《数据库系统》

竝

Chapter 1 关系模型与SQL

1.1 关系模型

1.1.1 结构

一张表对应一个**关系**(*relation*)，行对应**元组**(*tuple*)，列对应**属性**(*attribute*)。

所有属性值的取值集合称为**域**(*domain*)，属性是**原子的**(*atomic*)，即不可分割的。

> 一个关系是所有属性值笛卡尔积的子集。

1.1.2 码 *key*

- **超码(*superkey*)**：一个关系中可以唯一标识某个元组的若干属性集合，即任意两个元组的超码是不同的。若 (A) 是某个关系的超码，那么 (A, B) 也是。

  > 如对于一所学校的个人信息表，人名不能是超码，而学号一定是。

- **候选码(*candidate key*)**：任意真子集不是超码的超码，不含任何多余属性。

  > 最小的超码。

- **主码(*primary key*)**：管理员指定的候选码。
- **外码(*foreign key*)**：在某个关系 R_1 中作为关系 R_2 的主码，则称为 R_1 的外码。

1.1.3 关系运算

<基本>	表达式	含义
选择	$\sigma_p(r)$	返回关系 r 中满足关系式 p 的元组的关系
投影	$\prod_{A_1,A_2,\ldots,A_k}(r)$	返回关系 r 中属性为 $A_1, A_2, \ldots, A_k$ 的列并去重
集合并	$r \cup s$	将两个属性数相等且所有属性的域相同的两个关系合并为同一个关系，并去重
集合差	$r - s$	返回属于关系 r 却不出现在关系 s 中的元组的关系
笛卡尔积	$r \times s$	返回任意两个关系的元组组合（两个关系的属性应不相交，否则应重命名） 时间复杂度过高，应注意优化算法
重命名	$\rho_x(E)$	将 E 重命名为 x 并返回。

<拓展>	表达式	含义
集合交	$r \cap s$	取同时出现在两个关系中的元组，亦可写作 $r - (r - s)$
自然连接	$r \bowtie s$	取两个关系公共属性中具有相同属性值的元组进行拼接。
theta 连接	$r \bowtie_\theta s$	返回满足关系式 θ 的自然连接结果
除	$r \div s$	关系 s 的属性是关系 r 的子集，通常用于处理“for all”类型查询
赋值	$\leftarrow$	简化表达
广义投影	$\prod_{F_1,F_2,\ldots,F_k}(r)$	F_i 是涉及常量和属性的函数，可以对数值结果进行四则运算
聚集	$_{G_1,G_2,\ldots,G_n}\mathcal{G}_{F_1(A_1),F_2(A_2),\ldots,F_n(A_n)}(r)$	G_i 为用于分组的属性，F_i 是聚集函数，A_i 是属性名 常见聚集函数有 ***sum/max/min/avg/count/...***

1.2 SQL 关系定义

```sql
# 创建
create table table_name1(
    ID char(5),
    name varchar(5) not null,
    dept_name varchar(20),
    salary numeric(8, 2),
```

```
    primary key (ID),
    foreign key(dept_name) references table_name2,
    check (salary >= 0)             # check(P) 确保表达式 P 在该关系中的存在
)

# 更新
alter table table_name1 add A D     # 添加属性 A, D 是类型
alter table table_name1 drop A

# 删除
drop table table_name1;
delete from table_name1;
```

1.3 SQL 查询

执行顺序：*from → where → group (aggregate) → having → select → order by*

1.3.1 基本查询

- ***select***：支持含有四则运算的算术表达式，运算对象可以是常数或属性。可用`select distinct`来去重，也可用`select all`显式指明无需去重。默认不去重。用`select *`查询表中所有属性。
- ***from***：定义了所有列出关系上的笛卡尔积。若列出的多个关系有重名属性，必须以表名为前缀来区别。
- ***where***：支持逻辑连词`and or not`，也支持`between and`指定查询范围。支持比较运算符。

1.3.2 连接

- 内连接(*natural join*)：常用于`from`子句中的关系运算，将两个在同名属性上具有相同值的关系对应的操作大大简化。
    - `join ... using (...)`：只需要在指定属性上的取值匹配。
    - `join ... on P`：满足谓词 P 即匹配。
- 外连接(*out join*)：`left / right / full outer join`只保留出现在左边/右边/全部关系中的元组。

1.3.3 更名

为了引用简洁或便于区分，可在***from***子句中使用`as new_name`进行关系更名，***select***也支持更名。

1.3.4 元组显示次序

`order by att_name`，用`desc`表示降序，`asc`表示升序。默认升序。

> 例：`order by salary desc, name asc`表示按工资降序，工资相同则按姓名升序。

1.3.5 集合运算

`union, intersect, except` 分别对应集合中的并集、交集、差集运算。可以用来连接两条不同的查询语句。

三者都是自动去重的，若要保留重复项必须在后面加 `all`。

1.3.6 聚集函数

固有聚集函数：`avg / max / min / sum / count`

可以使用 `distinct` 关键字来去重（`count(*)` 时不行）

`group by` 进行分组聚集，对某个属性，将具有相同值的元组分在一个组中。省略该语句表示将整个关系视为一个分组。后面也可以用 `having` 子句对分组限定条件，效果等同 `where`。只有在形成分组后才起作用。`select` 对最后生成的分组进行筛选。

> 注意：需保证任何没有出现在 `group by` 子句中的属性，如果出现在 `select/having` 语句中，则必须在聚集函数中。

```sql
## 下述SQL语句返回表中平均工资大于 10000 的人名并去重。
select count(distinct name)
from table_name1
group by dept_name
having avg(salary) > 10000
```

1.3.7 空值

将涉及空值 *null* 的任何比较运算的结果视为 ***unknown***（*true* / *false* 外的第三种逻辑值），算术运算的结果视为 *null*。

`is unknown / is null` 可以用来测试是否为未知/空值。

除了 `count(*)` 外的所有聚集函数都忽略输出集合中的空值。同时规定空值的 `count` 运算值为 0，其他所有聚集运算在输入为空值的情况下返回一个空值。

1.3.8 嵌套子查询

子查询是嵌套在另一个查询中的 `select - from - where` 表达式。对于 `select / from / where` 后面任意关系可以出现的位置，都可以被替换为一个子查询。用 `in / not in` 表示某些属性是否在子查询返回的关系中。

来自外层查询中重命名的相关名称可以用在子查询中，称为**相关子查询**。

- **集合的比较：** `some + subquery`：存在某值即可；`all + subquery`：对所有值进行比较。
- **空关系测试：** `exists + subquery` 判断是否存在某关系。`not exists(B except A)` 判断关系 A 包含关系 B。
- **重复元组存在性测试：** `unique` 判断查询结果无重复元组。
- **from 中的子查询：**子查询不能使用子句中其他关系的相关变量，除非用 `lateral` 作为前缀。
- **with 子句：**定义临时关系，这个定义只对包含 `with` 子句的查询有效。

1.3 SQL 修改

```sql
# 插入
insert into ... values(e1, e2, ...)

# 删除
delete from person where name != '竝'

# 更新
update person
set salary = case
                when name = '竝' then salary * 2
                else salary / 2
             end
```

Chapter 2 数据库设计和 E-R 模型

2.1 E-R 模型(*Entity-Relationship*)

2.1.1 实体集和关系集

实体是独特的“对象”，可看似“类(*class*)”。实体由一系列**属性**描述，而实体集是由一系列相同性质或属性的实体组成的集合。在图中用**矩形**表示，主码用**下划线**标注。

关系集是相同类型关系的集合，是多个实体集上的数学关系。参与联系集的实体集数目称为联系集的**度**。在图中用**菱形**标识。关系集也可以附带属性，用**虚线**连接。

2.1.2 约束

- **映射基数**：在 E-R 图中**箭头**表示一，**直线**表示多。

  > 三元关系中箭头只能出现一次，否则会导致混淆。

- **参与约束**：若实体集中的每个实体都参与到关系集的至少一个关系中，则称为**完全参与(*total participation*)**，在 E-R 图中用**双直线**表示；反之则称为**部分参与(*partial participation*)**
- **弱实体集**：没有足够的属性以形成主码，与**强实体集**（含有主码）相对。弱实体集必须与强实体集有关系。弱实体集的分辨符由**下划虚线**标注，其主码由其分辨符加上标识实体集的主码构成。弱实体集与强实体集的联系需用**双线菱形**。

2.1.3 特化与概化

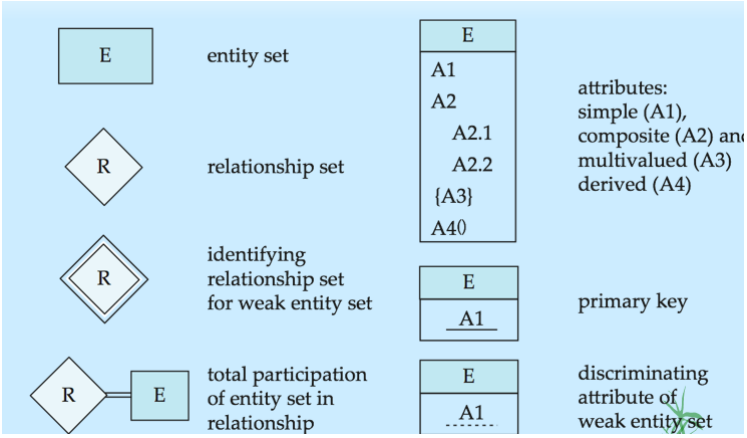
在实体内部进行分组的过程称为**特化**，自顶向下，实体内容不断细分，但继承上一级的属性。

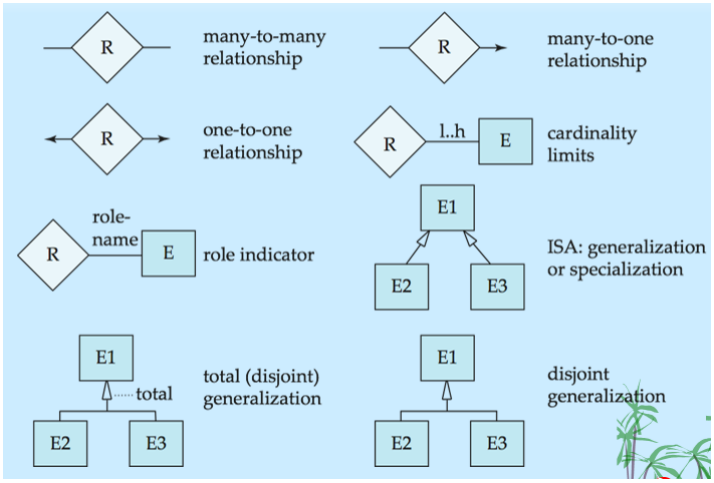
多个实体集根据共有的特征综合成一个较高层的实体集，称为**概化**，自底向上。

2.1.4 聚集

一个部分中所有的实体集均与同一个关系集相关联，则可把这部分合成为一个实体，在 E-R 图中用一个大方框框起来表示。

2.1.5 E-R 图





2.2 范式(*Normal Form*)

为了减少数据冗余，提出范式的概念。

2.2.1 分解

- 有损分解(***Lossy Decomposition***)：不能用分解后的几个关系重建原本的关系，反之为有损分解。
- 无损分解(***Lossless Decomposition***)：R 被分解为 (R_1, R_2) 且 $R = R_1 \cup R_2$ ，对于任何 R 上的关系 r 有 $r = \prod_{R_1}(r) \bowtie \prod_{R_2}(r)$

> 无损分解的充要条件：$R_1 \cap R_2$ 是 R_1 或 R_2 的超码。

2.1.2 第一范式

所有属性都是原子的(***atomic***)，即不可再细分的。

2.1.3 函数依赖

对于任何一个一个关系 R，若 $\alpha \subset R, \beta \subset R$，且由 R 中两个元组 t_1, t_2 中的 α 属性值相同可以得出其 β 属性值相同，则称 $\alpha \to \beta$ 是一个**函数依赖**，即 α 可以唯一标识/决定 β。若 R 中的每个元组都满足函数依赖，则称函数依赖在 R 上成立。

> 考虑数学中的“函数”，任何自变量对应唯一的因变量，有 $x \to f(x)$ 这一函数关系。

1. K 是 R 的超码 $\iff K \to R$
2. K 是 R 的候选码 $\iff K \to R$ 且不存在属性满足 $\alpha \subset K, \alpha \to R$（候选码是最小的超码）
3. 若 β 是 α 的子集，则一定有 $\alpha \to \beta$。称这样一个函数依赖是**平凡的(*trival*)**。

2.2.4 闭包

由给定的函数依赖 F 所能推导出的所有函数依赖构成的集合 F^+ 称为**闭包**。

- ***Armstrong* 公理**
    1. **自反律(*reflexivity rule*)**：若 $\beta \subseteq \alpha$，则 $\alpha \to \beta$
    2. **增补律(*augmentation rule*)**：若 $\alpha \to \beta$，则 $\gamma\alpha \to \gamma\beta$
    3. **传递律(*transitivity rule*)**：若 $\alpha \to \beta, \beta \to \gamma$，则 $\alpha \to \gamma$
    4. **合并律(*union rule*)**：若 $\alpha \to \beta, \alpha \to \gamma$，则 $\alpha \to \beta\gamma$
    5. **分解律(*decomposition rule*)**：若 $\alpha \to \beta\gamma$，则 $\alpha \to \beta, \alpha \to \gamma$
    6. **伪传递律(*pseudotransitivity rule*)**：若 $\alpha \to \beta, \lambda\beta \to \gamma$，则 $\lambda\alpha \to \gamma$
- **属性集闭包**：某属性所能唯一决定的属性的集合。若 $(\alpha \to \beta) \in F^+$，则 $\beta \in \alpha^+$。计算 α 的属性集闭包 α^+ 的伪代码如下：

```
result = {a};
while result is changed do
    for each x->y in F do
        begin
            if x in result then result += {y};
        end
return result
```

    1. 若 α^+ 包含 R 中所有属性，则 α 是超码。
    2. 若 $\beta \in \alpha^+$，则 $\alpha \to \beta$ 存在。
    3. **另一种计算函数依赖闭包的方法**：找出 R 上所有属性的属性集闭包 γ^+，对任意 $S \subseteq \gamma^+$，将 $\gamma \to S$ 加入结果集。

2.2.5 无关属性与最小/正则覆盖(*Canonical Cover*)

- **无关属性**：如果去除函数依赖中的某个属性不会改变这个函数依赖集的闭包，则称该属性是无关的。
    - α 中的 A 是否无关：计算 F 下 $(\alpha - A)^+$，若其包含 β 中所有属性，即 α 不需要 A 也能决定 β，则无关。
    - β 中 的 A 是否无关：计算 $(F - \{\alpha \to \beta\}) \cup (\alpha \to (\beta - A))$ 下 α^+，若其包含 A，即即使不需要在 β 中加入 A，也能通过各个函数依赖得到 $\alpha \to A$，则无关。
- **最小覆盖**：F 的最小覆盖 F_c 是个依赖集，使得 F 在逻辑与 F_c 是等价的（即左右可相互推导），此外最小覆盖必须满足

1. F_c 中任何函数依赖不含无关属性。
2. F_c 中任何函数依赖左半部分唯一，即任意 $\alpha_1 \to \beta_1, \alpha_2 \to \beta_2$ ，若 $\alpha_1 = \alpha_2$ ，必有 $\beta_1 = \beta_2$

- **寻找最小覆盖的方法**

1. 令 $F_c = F$
2. 利用合并律将所有 $\alpha \to \beta_1, \alpha \to \beta_2, \ldots$ 合并为 $\alpha \to \beta_1\beta_2 \ldots$
3. 在 F_c 中寻找一个具有无关属性的函数依赖，并删除该无关属性。
4. 重复上述步骤直至 F_c 不变。

2.2.6 保持依赖

令 F 为 R 上的一个函数依赖集，$R_1, R_2, \ldots, R_n$ 是 R 的分解，用 F_i 表示只包含 R_i 中出现的元素的函数依赖的集合，我们希望的结果是 $(F_1 \cup F_2 \cup \cdots \cup F_n)^+ = F^+$ ，具有这样性质的分解称为**保持依赖分解(*Dependency-Preserving Decomposition*)**。

2.2.7 *Boydd-Codd Normal Form*(BCNF)

若闭包中的**任何**一个函数依赖 $\alpha \to \beta$ 至少满足下面的一项：

1. $\alpha \to \beta$ 是**平凡的**
2. α 是 R 的一个**超码**

则称这个关系模式属于 BCNF。

- **判断方法**：计算 α^+ ，若 α^+ 既不包含所有元素（不是超码）又不包含 β（不平凡），则不是 BCNF。
- **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 的规则。不需要检查 F^+ 中所有的函数独立，可以证明如果 F 中没有违背 BCNF 规则的函数依赖，那么 F^+ 中也没有。

  > 这个方法不能用于检测 R 的分解

- **BCNF 分解**：可以证明这不仅是一个 BCNF 分解，还是一个无损分解。

  1. 令 $result = R, \quad done = false$
  2. 计算 F^+
  3. 如果 $result$ 中存在模式 R_i 违反了 BCNF 的规则，那么令 $\alpha \to \beta$ 为一个在 R_i 上成立的非平凡函数依赖且 $\alpha \cap \beta = \emptyset$ ，接着 $result = (result - R_i) \cup (R_i - \beta) \cup (\alpha, \beta)$ ；反之 $done = true$
  4. 重复 3 直至 $done = true$

  > 即找出 R_i 中使其不满足 BCNF 但在 R_i 中成立的那一个函数依赖 $\alpha \to \beta$ ，将 R_i 分解为 $(R_i - \beta) \cup (\alpha, \beta)$
  >
  > 例：$R = (A, B, C)$ 和 $F : \{A \to B\}$ ，可拆分为 $R_1 = (A, B), R_2 = (A, C)$

2.2.8 *3rd Normal Form*(3NF)

3NF 相比于 BCNF，对函数依赖放宽了要求，若闭包中的任何一个函数依赖 $\alpha \to \beta$ 至少满足下面的一项：

1. $\alpha \to \beta$ 是**平凡的**
2. α 是 R 的一个**超码**
3. $\beta - \alpha$ 中的每一个属性 A 都包含于 R 的一个候选码中（可以不是同一个）。

则称这个关系模式属于 3NF。

> 任何 BCNF 都属于 3NF。

- **判断方法**：不需要判断 F^+ 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 中的每一个属性是否包含在 R 的候选键中。
- **3NF 分解**

  1. 令 F_c 为 F 的正则覆盖，且 $i = 0$
  2. 对 F_c 中的每一个函数依赖 $\alpha \to \beta$，$i++, R_i = \alpha\beta$
  3. 若模式 $R_1, R_2, \dots, R_i$ 都不包含 R 的候选码，则 $i++, R_i = candidate\ key\ in\ R$
  4. 若模式 R_j 包含于另一个模式 R_k 中，则删除 R_j —— $R_j = R_i, i--$，直到没有模式可以删除。
  5. 返回 $(R_1, R_2, \dots, R_i)$

Chapter 3 XML

3.1 XML 的定义与基本结构

3.1.1 定义

XML(*Extensible Markup Language*) 是一种**可扩展标记语言**。与 HTML(*Hypertext Markup Language*) 不同的是，XML 中没有指定的标签集，用户可以根据需要添加新的标签，并单独指定如何处理标签显示，适合数据库之间的交互；而 HTML 中的标签数量有限。

3.1.2 基本结构

```
<university xmlns:yale = "http://www.yale.edu">
    ...
    <yale:course course_id = "CS-101">
        This course is being offered for the first time in 2009.
        <yale:title> Intro. to Computer Science</yale:title>
        <yale:dept_name> Comp. Sci. </yale:dept_name>
        <yale:credits> 4 </yale:credits>
    </yale:course>
    ...
</university>
```

- **标签(*Tag*)**：表明一种数据类型。

- 元素(*Element*)：被标签括起来的内容，以 `<tag> Element </tag>` 的形式表示。同种标签必须成对出现，用法跟大中小括号类似。文本与元素的混合使用是合法的。
- 根(*root*)：必须有一个独立的根元素来包含文档里的所有其他元素。
- 属性(*Attribute*)：一个元素可以有多个属性，但属性在给定标签中只能出现一次，以 `Name = Value` 的形式内嵌在起始标签中。
- 命名空间(*Namespace*)：XML 文档需在不同应用程序之间交换数据，相同的标签名称在不同的地方可能有不同的意义，为了避免混淆，在每个标签/属性的前面加上通用资源标识符（比如网址）`namespace:`。为了使用方便，在根元素中用属性 `xmlns` 声明命名空间。一份文档可以有多个命名空间。
- 字符数据：可以用 `<![CDATA[ string ]]>` 来表示正常的文本数据，而不会被解析为标签或其他结构。

3.2 XML 文档模式

3.2.1 文档类型定义(*Document Type Definition*, DTD)

是 XML 文档的一个可选部分，DTD 的主要为了对文档中出现的信息进行约束和类型限定——不限制基本类型，只限制元素中子元素的属性和出现。

基本语法结构如下：

```
<!DOCTYPE university [
        <!ELEMENT university ( (department|course|instructor)+)>
        <!ELEMENT department ( building, budget )>
        <!ATTLIST department
                dept_name   ID      #REQUIRED >
        <!ELEMENT building ( #PCDATA )>
        <!ELEMENT budget ( #PCDATA )>
        <!ELEMENT course ( title, credits )>
        <!ATTLIST course
                course_id   ID      #REQUIRED
                dept_name   IDREF   #REQUIRED
                instructors IDREFS  #IMPLIED >
        <!ELEMENT title ( #PCDATA )>
        <!ELEMENT credits ( #PCDATA )>
        <!ELEMENT instructor ( name, salary )>
        <!ATTLIST instructor
                IID         ID      #REQUIRED
                dept_name   IDREF   #REQUIRED >
        <!ELEMENT name ( #PCDATA )>
        <!ELEMENT salary ( #PCDATA )>
]>
```

- 子元素(*subelement*)：`<!ELEMENT element (subelement)>`
    - 子元素的声明

        - "元素名"
        - "#PCDATA"：表示文本数据
        - "EMPTY"：指这个元素没有内容。
        - "ANY"：对子元素无限制。
    - 子元素声明中的正则表达式
        - `|` 表示“或”，即可以相互替代。
        - `+` 表示“一个或多个”。
        - `*` 表示“零个或多个”。
        - `?` 用来指定一个可选的元素“零个或一个”。
- 属性(***Attribute***)：`<!ATTILIST element attribute_name attribute_type attribute_default>`
    - 属性类型声明
        - "CDATA"：字符数据。
        - "ID"：提供该元素的唯一标识符。每个元素**最多一个**属性作为 ID，并且必须是不同的值。
        - "IDREF"：是对一个元素的引用。IDREF 类型的属性必须包含出现在 ID 中的值。
        - "IDREFS"：允许以空格分开的引用列表。
    - 属性默认声明
        - "default value"：属性默认值
        - "#REQUIRED"：每个元素在该属性上必须指定一个值。
        - "#IMPLIED"：可以忽略这个属性。

> 局限性：元素和属性不能限定类型，且 ID 和 IDREF 缺乏类型限定，每个元素的 ID 属性值必须是唯一的

3.2.2 XML *Schema*

解决了 DTD 的缺点，更复杂，支持许多值的类型，包括 *integer*，*string* 等；允许用户自定义类型，可以是更复杂的类型；实现更多功能，包括唯一性和外键的约束。

与 DTD 不同，XML *Schema* 本身是用利用 XML *Schema* 定义的各种标签用 XML 的语法指定的。为了避免与用户定义标签冲突，在 XML *Schema* 标签上增加命名空间前缀 `xs:`，通过根元素的 `<xs: schema xmlns: xs = "...">` 声明与 XML *Schema* 命名空间关联。

3.3 XML 查询

3.3.1 XML *Tree Model*

一份 XML 文档被建为一棵树，其结点分为元素、属性、文本、命名空间、处理指令、注释以及文档（根）结点。

元素结点可以有子结点，即该元素的子元素或属性；相应的，除根元素外的每个结点都有一个父结点，即父元素。

根结点只有一个儿子，就是 XML 文档的根元素。

3.3.2 XPath

XPath 通过**路径表达式**指向 XML 文档的部分内容。

- **路径表达式**：由"/"隔开的定位步骤的一个序列（类似于文件路径），初始的"/"代表下一层标签之上的根。从左到右计算，其结果是一系列结点构成的集合。
- **特性**

    1. 用`@`访问**属性值**，如 `/u/@v` 返回 *u* 元素的 *v* 属性的所有值的集合。
    2. 谓词包含在方括号中，如 `/u[P]` 返回满足 P 的 *course* 元素。
    3. 布尔连接 *and*、*or* 可以在谓词中使用。
- ***count*() 函数**：`/instructor[count(./teaches/course)> 2]` 返回授课两门以上的老师。
- ***id*() 函数**：`id("values")` 返回属性类型为 ID 且值为 "*values*" 的结点，`/u/id(@ v)` 返回被 *u* 元素的 *v* 属性引用的所有元素。
- **操作符 "|"**：对表达式结果进行合并。
- "//"可以跳过多层结点，直奔目标。`/u//v` 找出 u 元素下**任意位置**上的所有 v 元素。不仅能往下查询，也可以往上查询，既可以搜索子结点，也可以搜索父结点（".."指定父结点）。
- ***doc*() 函数**：返回一个已命名文档的根。`doc(u.xml)/u/v` 返回 *u* 中的所有 *v* 元素。
- ***text()* 函数**：加在最后，去掉外围标签。

3.3.3 XQuery

- **基本语法 FLWOR**：`for ... let ... where ... order by ... return`

```
for $x in /university/course
let $courseId := $x/@course_id
order by $x/dept_name
where $x/credits > 3
return <course_id> { $courseId } </course id>
```

    上式返回所有学分大于 3 的 *course* 的 *course_id* 并按 *dept_name* 进行排序。

    `for` 类似于 SQL 中的 *from*；`let` 允许将 XPath 表达式结果赋值给变量名；`where, order by` 和 SQL 中的一样；`return` 允许构造 XML 形式的结果。
- ***Join* 连接**：在 *where* 后面用逻辑关系连接若干个条件
- **嵌套查询**：将一个 XQeury 语句当作 *return* 的结果用 `<tag> ... </tag>` 包起来
- **聚合**：聚合函数有一个默认名字空间前缀 `fn`，因此可以以 `fn: sum, fn: count` 的形式被无歧义调用。
- **排序**：可以用 *order by* 对结果排序，*descending* 降序 *asscending* 升序。

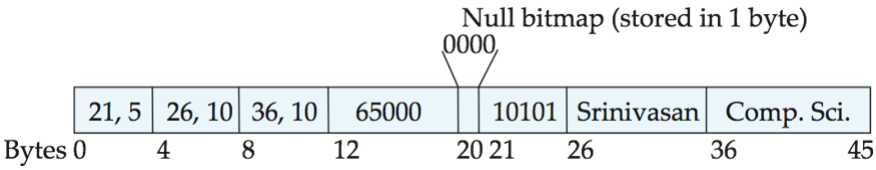
Chapter 4 索引与查询

4.1 文件结构与索引

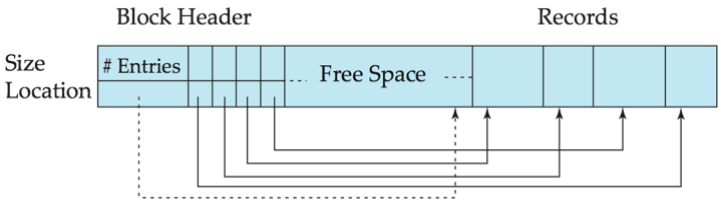
主要是 *B+Tree* 的考察。

4.1.1 文件结构

- **定长记录**：分配一定数量的字节作为文件头，被删除的记录形成称为**空闲列表**的链表。
- **变长记录**：（偏移量，长度）+ 定长部分 + 空位图 + 变长部分。空位图指明了记录的哪个属性是空值。



- **分槽页结构**



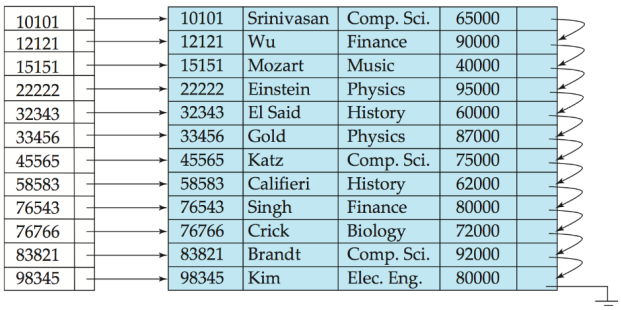
4.1.2 顺序索引

Search Key 搜索码，在文件中通过一个属性值查找一系列属性值。

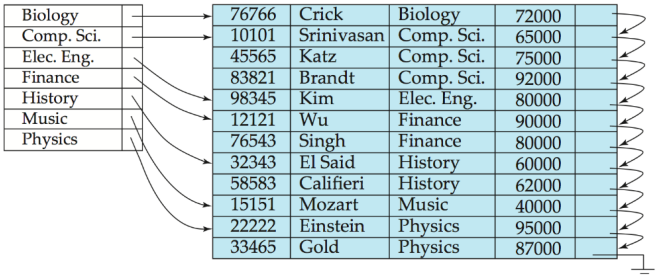
Index File 索引文件包含一系列的 *Search Key* 和 *Pointer*（两者的组合被称为 *Index Entry* 索引项），查询方式是通过 *Search Key* 在 *Index File* 中查询数据的地址(*Pointer*)，然后再从 *Data File* 中查询数据。

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* 为次关键字。

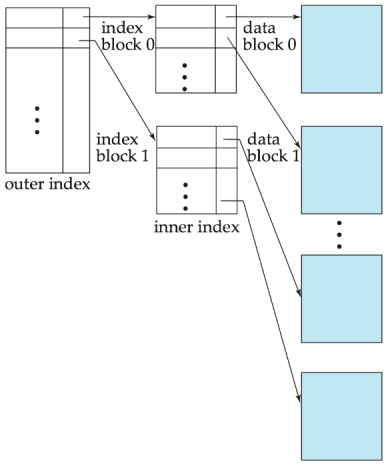
- ***Dense Index* 密集索引**：每一条记录都有对应的索引。



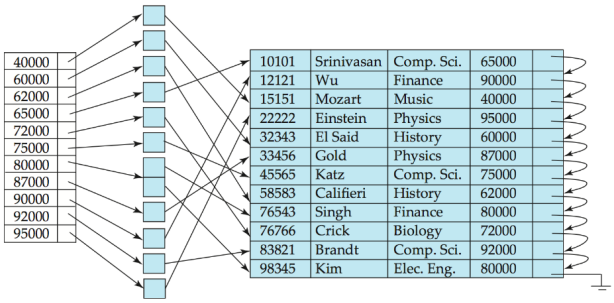
- ***Sparse Index* 稀疏索引**：只为搜索码的某些值建立索引项。只有当关系按搜索码排列顺序存储时才能使用稀疏索引。空间开销小，但索引速度较慢。定位包含所要查找记录的块是一个不错的 *trade-off* 。



- ***Multilevel Index* 多级索引**：和多级页表差不多？

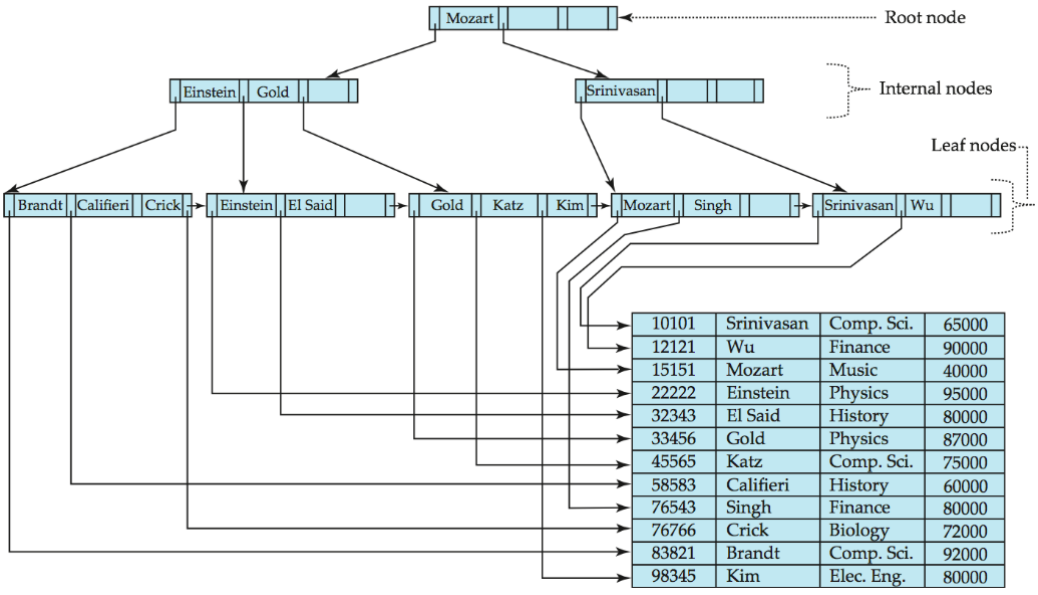


- ***Secondary Index* 辅助索引**：必须是稠密索引。通过一个包含所有文件指针的 *bucket* 来寻找所指向的地方。



4.1.3 B+ 树索引

非叶结点有 $\lceil n/2 \rceil \sim n$ 个子结点，叶结点有 $\lceil (n-1)/2 \rceil \sim n-1$ 个值。



- **查询**：$O(\log N)$ 。非叶结点中，先找到第一个 $K_i \geq V$ 的搜索码，若 $K_i = V$ ，则走 P_{i+1} 指向的结点；反之，走 P_i 指向的结点。若不存在该码，则走最后一个非空指针指向的结点。叶结点中，找到第一个 $K_i = V$ 的码，返回索引，若不存在，则返回空。
- **增加**：增加后，若结点中值的数量超过上限，则分裂，并一路向上更新索引。
- **删除**
    - 若删除后结点中值的数量未破坏 B+ 树的规则，则直接删除该值，并更新所有祖先结点。
    - 反之，若兄弟够借，则将兄弟的值拿一个过来（相当于是把兄弟结点的那个值删除了）；否则，从旁边找一个叶节点来合并出新的非叶节点。并更新祖先结点。
    - 在更新祖先结点的同时若破坏了非叶结点的规则，同理操作。
- **高度**
    - **最小情况**：所有叶结点都存满搜索码，此时 $h = \lceil \log_{N/2} M \rceil$
    - **最大情况**：所有叶结点都半满，此时 $h = \lceil \log_{N/2} \frac{M}{2} \rceil + 1$

4.2 查询处理 *Query Processing*

4.2.1 查询步骤

语法分析与翻译、优化、执行。

4.2.2 查询代价

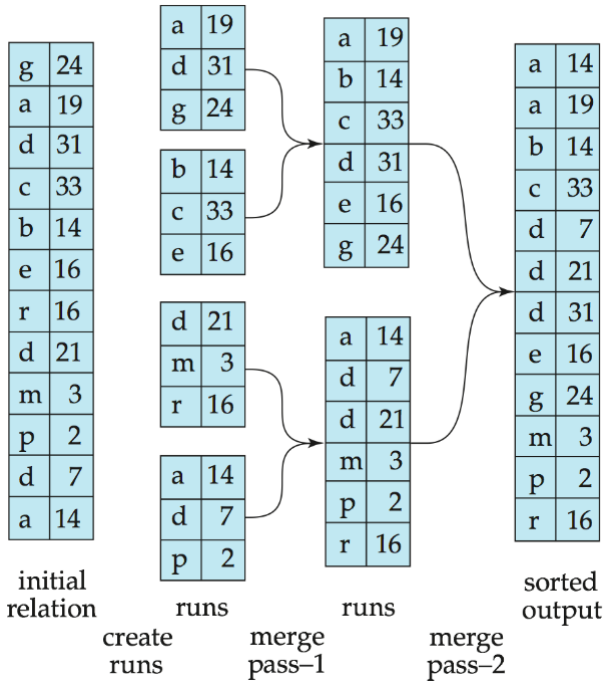
一次传输 B 个块以及执行 S 次磁盘搜索的操作将消耗 $B \cdot t_T + S \cdot t_S$ (s)，其中 t_T 为传输一个磁盘块数据的时间，t_S 为磁盘块平均访问时间。主要的代价来源还是磁盘的 *access*。

4.2.3 选择运算

- **线性搜索**
    - 扫描每一个文件块，对所有记录进行测试，判断其是否满足选择条件，此时 $cost = B_r \cdot t_T + t_S$
    - 若通过**码属性等值比较**，则平均代价 $cost = (B_r/2) \cdot t_T + t_S$ ，最差代价同上。二分法不起作用。
- **B+树主索引**
    - **码属性等值比较**：只需搜索一条记录。索引查找穿越 B^+ 树的高度，再加上一次 *I/O* 来取记录，每个 *I/O* 需要一次搜索和一次块传输，代价 $cost = (h_i + 1) \cdot (t_T + t_S)$
    - **非码属性等值比较**：需要搜索多条记录，每一层对第一个快进行一次搜索。$cost = h_i \cdot (t_T + t_S) + B \cdot t_T$
- **B+树辅助索引**
    - **码属性等值比较**：此时能检索到唯一记录，与 A2 类似，$cost = (h_i + 1) \cdot (t_T + t_S)$
    - **非码属性等值比较**：可能检索到多条记录，有时候会非常耗时，$cost = (h_i + n) \cdot (t_T + t_S)$

4.2.4 排序

- **外部排序归并算法**：M 表示内存缓冲区中可以用于排序的块数，b_r 表示记录的块的数量。每次归并取 $M-1$ 个块进行输入，1 个块进行输出。
- **归并趟数**：$\lceil \log_{M-1}(b_r/M) \rceil$
- **创建和每次归并所需磁盘块传输次数**：$2b_r$
- **磁盘块传输总数**：$b_r(2\lceil \log_{M-1}(b_r/M) \rceil + 1)$



4.2.5 连接运算

$r \bowtie s$，其中 r 为外层关系，s 为内层关系。b_r, b_s 分别为两个关系所需块数，n_r, n_s 分别为两个关系所含元组数。

连接方式	磁盘传输次数	磁盘搜索次数	备注
Nested-loop Join	$n_r \cdot b_s + b_r$	$n_r + b_r$	/
Block Nested-loop Join	$\lceil b_r/(M-2) \cdot b_s + b_r \rceil$	$\lceil 2b_r/(M-2) \rceil$	M 是内存可以容纳的 *block* 数量
Index Nested-loop Join	/	/	$cost = b_r(t_T + t_S) + n_r * c$ c 是使用连接条件对关系 s 进行单次选择操作的代价。
Merge Join	$b_r + b_s$	$\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil$	b_b 为为每个关系分配的缓冲块数。
Hash Join	$3(b_r + b_s) + 4n_h$	$2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)$	n_h 为元组的划分数，b_b 为为每个关系分配的缓冲块数。

4.3 *Query Optimization*

4.3.1 等价关系表达式

构造前缀树，利用等价规则进行转换，并选取代价最低的表达式。

- 选择：$\sigma_{\theta_1\wedge\theta_2}(E)=\sigma_{\theta_1}(\sigma_{\theta_2}(E))=\sigma_{\theta_2}(\sigma_{\theta_1}(E))$
- 投影：$\prod_{L_1}(\prod_{L_2}(\dots(E)))=\prod_{L_1}(E)$
- 自然连接：$E_1\bowtie(E_2\bowtie E_3)=(E_1\bowtie E_2)\bowtie E_3$
- ***theta*** 连接：$(E_1\bowtie_{\theta_1}E_2)\bowtie_{\theta_2\wedge\theta_3}E_3=E_1\bowtie_{\theta_1\wedge\theta_3}(E_2\bowtie_{\theta_2}E_3)$
- 选择与笛卡尔积、**theta** 连接
    - $\sigma_\theta(E_1\times E_2)=E_1\bowtie_\theta E_2=E_2\bowtie_\theta E_1$
    - $\sigma_{\theta_1}(E_1\bowtie_{\theta_2}E_2)=E_1\bowtie_{\theta_1\wedge\theta_2}E_2$
- 选择分配律
    - 当 θ_1 只涉及 E_1 中的属性，$\sigma_{\theta_1}(E_1\bowtie_{\theta_2}E_2)=\sigma_{\theta_1}(E_1)\bowtie_{\theta_2}E_2$
    - 当 θ_1,θ_2 分别都只涉及 E_1,E_2 中的属性，$\sigma_{\theta_1\wedge\theta_2}(E_1\bowtie_\theta E_2)=\sigma_{\theta_1}(E_1)\bowtie_\theta\sigma_{\theta_2}(E_2)$
- 投影分配律：当 θ 只涉及在 $L_1\cup L_2$ 中的属性，$\prod_{L_1\cup L_2}(E_1\bowtie_\theta L_2)=(\prod_{L_1}(E_1))\bowtie_\theta(\prod_{L_2}(E_2))$
- 集合的并、交运算满足交换律与结合律，选择对并、交、差运算有分配律。
- 投影对并运算有分配律。

> 连接次序：先从连接后数据量最小的开始。

4.3.2 结果集大小估计

在关系 r 中声明以下变量：n_r 表示关系中的元组数，b_r 表示关系中元组的磁盘块数，l_r 表示关系中每个元组的字节数，f_r 表示一个磁盘块中能够容纳的元组的个数，$V(A,r)$ 表示关系中属性 A 中出现的非重复值个数。

> 当关系中的元组都存储在一个文件中的时候，$b_r=\lceil n_r/f_r\rceil$

- 选择估计（均匀分布）
    - 从关系中选择属性 A 为某个值 x：$size=n_r/V(A,r)$
    - 选择属性 A 中小于某个值 x 的元组
        - 若 $x<\min(A,r)$，则 $size=0$
        - 反之，$size=n_r\cdot\frac{x-\min(A,r)}{\max(A,r)-\min(A,r)}$
    - 选择属性 A 中大于某个值 x 的结果集大小恰好与上条对称。
- 复杂选择估计

  假设 s_i 是满足条件 θ_i 的元组的个数，因此关系中的一个元组满足选择条件 θ_i 的概率为 s_i/n_r。
    - 合取：$size(\sigma_{\theta_1\wedge\theta_2\wedge\dots\wedge\theta_n}(r))=n_r\cdot\frac{s_1\cdot s_2\cdot\dots\cdot s_n}{n_r^n}$
    - 析取：$size(\sigma_{\theta_1\vee\theta_2\vee\dots\vee\theta_n}(r))=n_r\cdot(1-(1-\frac{s_1}{n_r})-(1-\frac{s_2}{n_r})\dots(1-\frac{s_n}{n_r}))$
    - 取反：$size(\sigma_{\neg\theta}(r))=n_r-sizeof(s)$
- 连接估计 $r\bowtie s$
    - 若两个关系无共同属性，则 $size(r\bowtie s)=n_r\cdot n_s$
    - 若两个关系的共同属性是 r 的码，则 $size(r\bowtie s)\le n_s$

- 若两个关系的共同属性是 s 到 r 的外码，则 $size(r \bowtie s) = n_s$
- 一般情况下，$size(r \bowtie s) = (n_r \cdot n_s)/(\max(V(A,r), V(A,s)))$

- 其他估计
  - 投影：$size(\prod) = V(A,r)$
  - 聚合：$size(\mathcal{G}) = V(A,r)$
  - 集合：转换为合取、析取
  - 外连接
    - 左外连接：$size = size(r \bowtie s) + n_r$
    - 右外连接：$size = size(r \bowtie s) + n_s$
    - 全外连接：$size = size(r \bowtie s) + n_r + n_s$

Chapter 5 事务

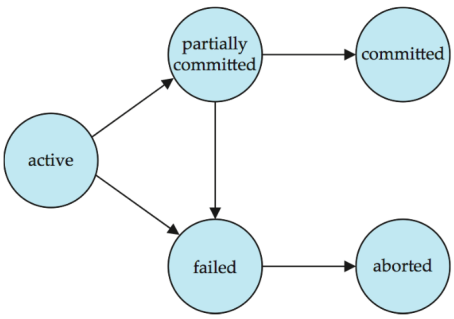
5.1 事务

5.1.1 事务概念

事务是访问并可能更新各种数据项的一个程序执行单元。需解决两大问题：

1. 各种故障，如硬件故障和系统崩溃。
2. 多个事务的并发执行。

- 性质 **ACID**
  - **原子性(*Atomicity*)**：事务中的所有步骤只能 *commit* 或者回滚 *rollback*
  - **一致性(*Consistency*)**：单独执行事务可以保持数据库的一致性
  - **隔离性(*Isolation*)**：事务在并行执行的时候不能感知到其他事务正在执行，中间结果对于其他并发执行的事务是隐藏的。
  - **持续性(*Durability*)**：更新之后哪怕软硬件出了问题，更新的数据也必须存在。
- 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。



5.1.2 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。

- **调度(*Schedule*)**：一系列用于指定并发事务的执行顺序的指令。需包含事务中的所有指令，且保证单个事务中指令的相对顺序。
    - **事务的最后一步**
        - 成功执行：***commit instruction***
        - 执行失败：***abort instruction***
    - **串行调度(*Serial Schedule*)**：一个事务调度完成之后再进行下一个。
    - **等价调度(*Equivalent Schedule*)**：改变处理的顺序但是和原来等价。
    - **冲突等价(*Conflict Equivalent*)**：如果调度 S 可以通过一系列非冲突指令交换转换为 S'，则称 S 与 S' 等价。

        > 对同一数据项而言，当且仅当两个指令都是读时不引发冲突，其执行顺序无关紧要
        >
        > 若两条指令针对的数据项不同，则读写都不冲突。

    - **冲突可串行化(*Conflict Serializability*)**：当 S 与一个串行调度等价时，称 S 是冲突可串行化的。
    - **优先图(*Precedence graph*)**：顶点集由所有参与调度的事务组成，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，则有边 $T_i \rightarrow T_j$ 。一个调度是**冲突可串行化**的当且仅当优先图是**无环图**。对于无环图，可以使用拓扑排序获得一个合适的执行顺序。
- **可恢复调度(*Recoverable Schedules*)**：若 T_j 读取了先前由 T_i 所写的数据项，则 T_i 必须在 T_j *commit* 之前就 *commit* 。否则，若 T_i 出现故障，则不能正确恢复。
    - **级联回滚(*Cascading Rollbacks*)**：即使一个调度是可恢复的，但要从某事务的故障中正确恢复，可能需要回滚若干事务。
    - **无级联调度(*Cascadeless Schedules*)**：为了避免级联回滚发生，应满足：若 T_j 读取了先前由 T_i 所写的数据项，则 T_i 必须在 T_j 的读操作之前提交。

5.2 并发控制

当多个事务并发执行时，为了保持隔离性，采用并发控制这一机制。

> 17-18 18-19 两张卷子都没怎么涉及……不知道是不是重点。

5.2.1 基于锁的协议

锁是一种控制并发访问同一数据项的机制。两种锁：

1. **共享的*(shared(S)-mode lock)***：若事务获得了数据项上的该锁，则事务对该数据项**可读但不可写**。
2. **排他的*(exclusive(X)-mode lock)***：若事务获得了数据项上的该锁，则事务对该数据项**可读可写**。

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。

如果一个锁的申请没有被批准，就会产生一个请求事务并等待，直到所有冲突的锁被释放。

- **特殊情况**

    1. **死锁(*dead lock*)**：两个事务中的锁互相等待造成事务无法执行。
    2. **饥荒*(Starvation)***：如某事务申请一个数据项的 *X* 锁，而别的事务不断申请该数据项的 *S* 锁。

- **两阶段封锁协议**：要求每个事务分两个阶段提出加锁和解锁申请。**无法解决死锁的问题**。

    1. **增长阶段(*growing phase*)**：只能获得锁，不能释放。
    2. **缩减阶段(*shrinking phase*)**：只能释放锁，不能获得。

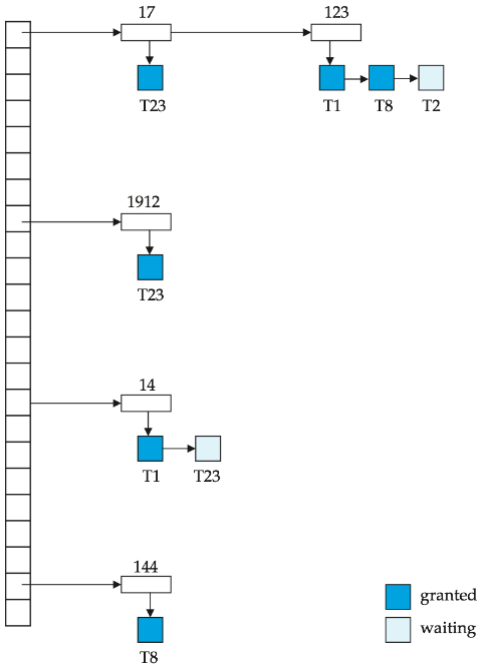
    最初，事务处于增长阶段，根据需要获得锁。一旦其释放了锁，就进入缩减阶段，不能发出加锁请求。

    - **严格两阶段封锁协议(*strict two-phase locking protocol*)**：要求事务持有的 *X* 锁必须在事务提交之后方可释放。解决级联回滚的问题。
    - **强两阶段封锁协议(*rigorous two-phase locking protocol*)**：要求事务提交前不得释放任何锁。

- **锁转换(*Lock Conversions*)**：提供了一种将 *S* 锁升级为 *X* 锁，*X* 锁降级为 *S* 锁的机制。只能在增长阶段升级，缩减阶段降级。

- **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，且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。

    - 使用一个以数据项名称为索引的散列表 *Lock table* 来查找数据项。
    - 深色框代表上锁，浅色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。



    - **请求处理**

        - 新的加锁请求置于队列末端，当且仅当和其他锁兼容时会被授予。
        - 收到解锁的请求后，*Lock Manager* 会删除对应的请求，并逐一检查后面的请求是否可以被授权。

- 如果一个事务终止了，所有该事务正在等待加锁的请求都会被删除。
- *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。

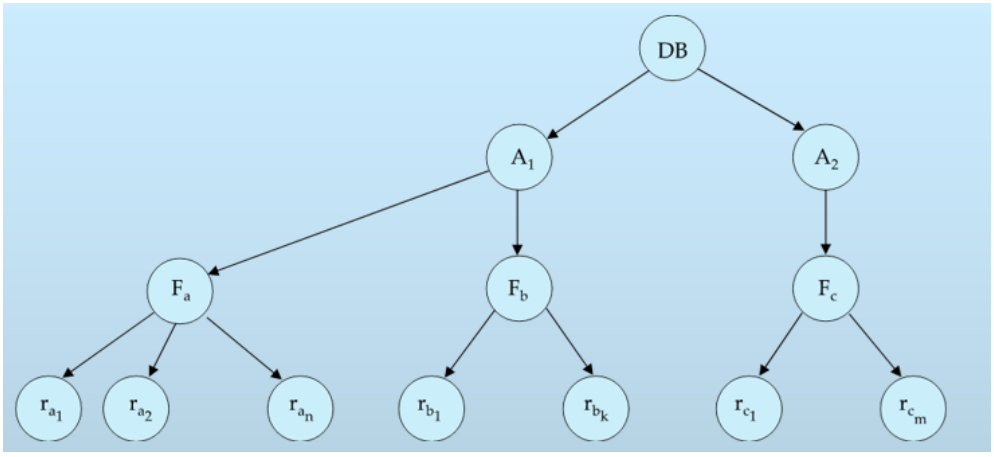
5.2.2 死锁处理

- 死锁预防
  1. 通过对加锁请求进行排序或要求同时获得所有的锁来保证不会发生循环等待。
  2. 使用抢占与事务回滚。
     - ***wait-die***：非抢占。老的事务可以等待，新的事务直接回滚。
     - ***wound-wait***：抢占。老的事务强制让新的事务回滚而不等待其释放，新的事务等待。
     - ***lock timeout***：事务等待时间有限，超时则回滚。容易实现，但会导致饥饿。
- 死锁检测：等待图：所有的事务表示图中的点，如果事务 *i* 需要 *j* 释放一个数据项，则在图中画一条点 *i* 到点 *j* 的有向边，如果图中有环，说明系统存在一个死锁。
- 死锁恢复
  - 彻底回滚(***total rollback***)：将事务终止之后重启
  - 部分回滚(***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

5.2.3 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。



- 意向锁：在一个结点显式加锁前，其全部祖先均加上意向锁。
  - ***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.

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

5.3 恢复系统

5.3.1 日志记录

更新日志记录描述一次数据库写操作，字段如下：

1. 事务标识：是执行 ***write*** 操作的事务的唯一标识。
2. 数据项标识：是所写数据项的唯一标识，通常是数据项在磁盘上的位置。
3. 新值/旧值：是数据项写前/后值。

表示为 $< T_i, X_j, V_1, V_2 >$，表明事务 T_i 对数据项 X_j 执行一次写操作，写前值为 V_1，写后值为 V_2。

- 数据库修改
  - 延迟修改(***deferred-modification***)：事务直到提交的那一刻才对数据库进行修改。
  - 立即修改(***immediate-modification***)：在事务活跃期间便对数据库进行修改。
- 检查点：$< checkpoint\ L >$ 用来提高恢复过程的效率，其中 L 是执行检查点时正活跃的事务的列表。对于检查点前的事务不执行操作，其后的事务根据有无 *commit* 或 *abort* 来执行重做***(redo)***或撤销***(undo)***操作。在崩溃发生后，只需反向搜索日志，找到最后一条检查点记录即可。

> 重做：将数据项的值设为新值。　　撤销：将数据项的值设为旧值。

5.3.2 恢复算法

- 正常事务回滚：反向扫描日志，对每一条记录执行撤销操作，并往日志中写一个只读记录 $< T_i, X_j, V_1 >$，一旦出现 $< T_i, start >$ 记录，就停止扫描，并往日志中写一个 $< T_i, abort >$
- 故障恢复
  - 重做阶段：从最后一个检查点开始正向扫描，将要回滚的事务的列表 *undo-list* 设定为 L 列表，遇到正常记录或 *redo-only* 记录就重做，遇到 $< T_i, start >$ 时将 T_i 加入 *undo-list*，遇到 $< T_i, abort >$ 或 $< T_i, commit >$ 时将 T_i 从 *undo-list* 中删除。
  - 撤销阶段：反向扫描日志，将在 *undo-list* 中的事务进行撤销，并往日志中写一个只读记录 $< T_i, X_j, V_1 >$，遇到列表中事务 T_i 的 $< T_i, start >$ 记录时往日志里写入 $< T_i, abort >$ 记录并将 T_i 从 *undo-list* 中删除。直至 *undo-list* 变为空表。

5.4 *ARIES*

1. 使用**日志顺序号(*Log Sequence Number, LSN*)**来标识日志记录，并将其存储在数据库页中。
2. 支持**物理逻辑** ***redo*** 操作。
3. 使用**脏页表(*dirty page table*)**来最大限度减少不必要的重做。
4. 使用模糊检查点机制，只记录脏页信息和相关信息。

5.4.1 数据结构

- ***LSN***：用于逻辑标识记录，是线性增长的。
- **页日志序号(*PageLSN*)**：更新操作将其 *LSN* 存储在该页的 *PageLSN* 域中，在恢复的撤销阶段 *LSN* 小于或等于 *PageLSN* 值的日志记录将不会执行。
- ***PreLSN***：指向同一事务的前一个日志记录的 *LSN*
- **脏页表(*dirty page table*)**：为每一页保存 *PageLSN* 和 *RecLSN* 的字段。当日志首次插入脏页表时，*RecLSN* 修改为日志的当前末尾，只要页被写入磁盘，就从脏页表中移除。

5.4.2 恢复算法

- **分析阶段**：找到最后的完整检查点日志记录，并读入脏页表，确定 *redo* 起点 *RedoLSN*（脏6TV页表中页的 *RecLSN* 最小值）。将要回滚的事务的列表 *undo-list* 设定为 L 列表，继续正向扫描，遇到不在 *undo-list* 中的事务的日志记录就将该事务加入，遇到 *end* 记录就将该事务从 *undo-list* 中删除。同时更新脏页表，将不在脏页表中的页加入并设置 *RecLSN* 为该日志记录的 *LSN*。
- **重做阶段**：从 *RedoLSN* 开始正向扫描，跳过不在脏页表中或 *LSN* 小于脏页表中该页 *RecLSN* 的日志记录，其余的全部重做。
- **撤销阶段**：反向扫描，对 *undo-list* 中的所有事务进行撤销，并往日志中写一个只读记录 $< T_i, X_j, V_1 >$，遇到列表中事务 T_i 的 $< T_i, start >$ 记录时往日志里写入 $< T_i, abort >$ 记录并将 T_i 从 *undo-list* 中删除。直至 *undo-list* 变为空表。