第1章 引言

基本概念

数据(Data)：数据库中存储的基本对象 数据库(Database,简称DB)：长期储存在计算机内、有组织的、可共享的大量数据集合 数据库管理系统(DataBase-Mangement System, DBMS)：由一个互相关联的数据集合和一组用以访问这些数据的软件组成

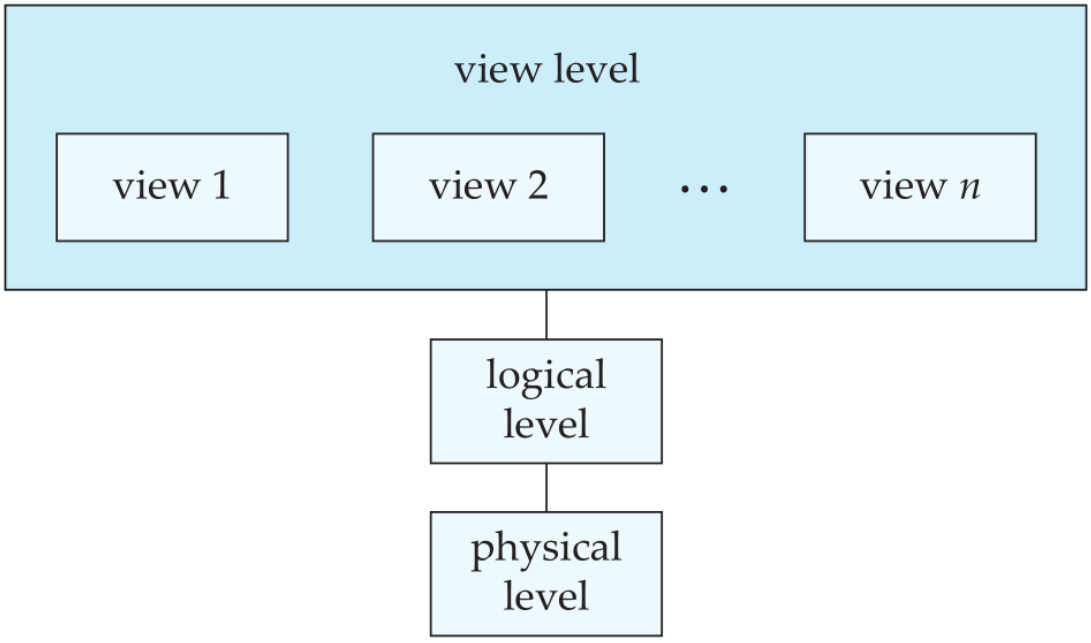
数据库系统的目标(文件处理系统的弊端)

1. 数据的冗余和不太一致(data redundancy and inconsistency)
2. 数据访问困难(difficulty in accessing data)
3. 数据孤立(data isolation)
4. 完整性问题(integrity problem)
5. 原子性问题(atomicity problem)
6. 并发访问异常(concurrent-access anomaly)
7. 安全性问题(security problem)

数据视图

数据抽象

1. 物理层(physical level)：描述数据实际上是怎么存储的，含数据结构
2. 逻辑层(logical level)：描述数据库中存储什么数据以及这些数据间存在什么关系
3. 视图层(view level)：最高层的抽象，描述整个数据库的某个部分



实例和模式

实例(instance)：特定时刻存储在数据库中的信息的集合 模式(schema)：数据库的总体设计

物理数据独立(physical data independence)：修改物理层不影响逻辑层 逻辑数据独立(logical data dependency)：修改应用层不影响逻辑层

数据模型(data model)

数据模型是一个描述了数据、数据联系、数据语义以及一致性约束的概念工具的集合。提供了一种描述物理层、逻辑层以及视图层数据库设计的方式 分类：

1. 关系模型(relational model)：用表的集合来表示数据与数据之间的关系
2. 实体-联系模型(entity-relationship, ER)
3. 基于对象的数据模型(object-based data model)：可以看成是E-R模型模型增添了封装、方法和对象标识等概念
4. 半结构化数据模型(semistructured data model)：允许相同类型的数据项含有不同属性集的数据定义。

数据库语言

1. 数据操纵语言(Data-Mainpulation Language, DML)：访问或操纵数据
    - 分类
        - 过程式DML(procedural DML)
        - 声明式DML(declarative DML, nonprocedural)：与上一个不同的是，只需说明需要什么数据，而不用说明如何获取这些数据，如SQL
    - DDL作为输入，输出放在数据字典(data dictionary)，数据字典可以看作一张特殊的表，数据字典包含了元数据(metadata)，元数据是关于的数据的数据
2. 数据定义语言(Data-Definition Language, DDL)：定义数据库的模式
    - 一致性约束
    1. 域约束(domain constraint)：某个范围取值
    2. 参照完整性(referential integrity)
    3. 断言(assertion)：数据库需要时刻满足的某一条件
    4. 授权(authorization)

数据存储和查询

1. 存储管理器(Storage Management)
    1. 权限及完整性管理器(authorization and integrity manager)
    2. 事务管理器(transaction manager)：故障也能保证一致
        - 并发控制管理器(concurrency-control manager)
        - 恢复管理器(recovery manager)：满足atomicity 和 durability
    3. 文件管理器(file manager)
    4. 缓冲区管理器(buffer manager)
    5. 数据文件(data file)
    6. 数据字典(data dicitonary)：储存关于数据库结构的元数据
    7. 索引(index)
2. 查询处理器
    1. DDL解释器
    2. DML编译器
    3. 查询执行引擎(query evaluation engine)

事务(transaction)

定义：数据库应用中完成单一逻辑功能的操作集合 见14章

用户与管理员

- 无知的用户(Naive users)： 只使用先前写好的应用程序
- 老练的用户(sophisticated user)：使用数据库查询语言或数据分析程序
- 应用程序员(application programmer)：编写应用程序
- 专门的用户(specialized users)：编写特定的数据库应用，如知识库、专家系统
- 数据库管理员(database adminstrator, DBA)

第一部分 关系数据库

第2章 关系模型

关系数据库

关系数据库基于关系模型，使用一系列表来表达数据以及这些数据之间的关系 关系：表 元组(tuple)：行 属性(attribute)：列

码(key)

- 超码(superkey)：唯一标识一个元组
- 候选码(candicate key)：最小超码，其真子集不为超码
- 主码(primary key)：候选码之一
- 外码(foreign key)：另一个关系模式的主码 r1有r2的主码R2，R2在r1上称作参照r2的外码。关系r1也被称作外码依赖的参照关系(referencing relation)，r2叫做外码的被参照关系(referenced relation)

第6章 形式化关系查询

关系代数

关系代数是一种过程化的查询语言。

1. 选择(select)运算 形式：$\sigma_{condition}(table)$ 条件：`=`, `≠`, `<`, `≤`, `≥`, `^`, `v`, `¬`
2. 投影(project)运算 形式：$\Pi_{col_1,...}(table)$
3. 并运算 形式：$()\bigcup()$
4. 差集(set-difference) 形式：$() - ()$
5. 笛卡尔乘积(Cartesian product) 形式：$tabel1 \times table2$
6. 更名运算 形式：$\rho_x(E)$ 表示将E更名为x，并返回E，若E是n元的，$\rho_{x(A_1,..A_n)}(E)$则表示将各列更名为A_i 例子：$\Pi_{t.c}(\sigma_{t.c<d.c}(t \times \rho_d(t))$，选择非最大值
7. 附加的关系代数运算
    1. 交集$\bigcap$，$r \bigcap s = r - (r - s)$
    2. 自然连接$\bowtie$
    3. 赋值运算符$\leftarrow$，$tmp1 \leftarrow R \times S$
    4. 外连接：中文课本P132
        - 左连接
        - 右连接
        - 全连接
8. 扩展的关系代数运算
    1. 广义投影：选择的col可以与某个运算结合，如$\Pi_{c_1,c_2+2}(t)$
    2. 聚集G(符号类似G) $_{group\_col}G_{sum(col)}(table)$，还有`count_distinct`

第二部分 数据库设计

第7章 数据库设计和E-R模型

基本概念

1. 实体集(entity set)：相同类型的实体集合
    - 实体(entity)：现实世界中可区别于所有其他对象的一个“事物”或“对象”。
    - 实体集的外延(extension)：实体集的实体的实际集合
2. 联系集：相同类型联系的集合 联系(relationship)：多个实体之间的相互联系
3. 属性：将实体集映射到域的函数
    - 域(domain)(值集, value set)：属性可取值的集合
    - 分类
        - 简单(simple),复合(composite)
        - 单值(simgle-valued), 多值(multivalued)：一对一，多对一
        - 派生(derived)：由其他属性推导得到的
4. 度(degree)：参与联系集的实体的数量

约束

映射基数(mapping cardinality)（基数比率）

表示一个实体通过联系集能关联到的实体的个数

1. one-to-one
2. one-to-many
3. many-to-one
4. many-to-many

参与(participation)约束

实体参与联系的比例

- Total participation
- Partial participation

实体-联系 图 (E-R diagram)

基本结构

1. 矩形：实体
2. 菱形：联系集
3. 矩形下部分：属性，主码划横线
4. 线段：将实体集连接到联系集
5. 虚线：将联系集的描述属性(descriptive attribute)连接到联系集
6. 双线：实体在联系集的参与度-全部参与
7. 双菱形：连接到弱实体集的标志性联系集

其他内容：中文课本P172

映射基数

1. 箭头指向“一”

2. 横线指向“多”
3. 横线上可设置"min..max"，*表示任意 注：双线表示全部参与

复杂的属性

+------------------+ |Name | +------------------+ |Simple | |Composite | | a1 | | a2 | |{Multivalued}| |DeriveValue()| +------------------+

角色**(role)**

定义：实体在联系中扮演的功能称为实体的角色 在矩形和菱形之间的连线上标注角色名称

弱实体集

弱实体集(week entity set)：没有足够的属性以形成主码的实体集

- 弱实体集必须与另一个称为标识(identifying)或属主实体集(owner entity set)关联在一起才能有意义
- 弱实体集与其标识实体集相连的联系称为标识性联系(identifying relationship)
- 分辨符(disciminator)：区分弱实体集中的实体，E-R图中用下虚线表示
- 主码：标注的主码和其分辨符
- 弱实体集在标志性联系中是全部参与的

强实体集(strong entity set)：有主码的实体集

转为关系模式

多值属性在关系模式中被分解成多个简单的单值属性。

1. 强实体集的表：
    - 属性：各个属性
    - 主码：原来的主码
2. 弱实体集的表：
    - 属性：标识的主码、弱实体集的属性（这就导致了弱实体集的联系集的表一般是冗余的，所以一般不用弱实体集的联系集的表）
    - 主码：标识的主码和弱实体的分辨符
3. 联系集的表：
    - 属性：参与集的主码、联系集的描述属性
    - 主码（针对二元联系集）：
        1. 多对多：参与集的主码（此法也适合n元的多对多）
        2. 一对一：任何一个参与集的主码即可
        3. 一对多（多对一）：“多”的一方的主码（此法也适合n元的多对一或一对多）

模式的合并：实体集A到实体集B的联系集AB是多对一（或一对一）的联系，且实体集A是全部参与，那么可以将A和AB简化为A‘，A’比A多了B的主码（作为外码）。如果A不是全部参与的话，这样做合并会导致空值（B主码那一列）的出现。

实体-联系设计问题

1. 用实体集还是用属性
2. 用实体集还是联系
3. 二元还是n元联系：n元可以转二元

扩展的**E-R**特性

特化与概化

特化(specialization)：继承 概化(generalization)：与特化相反的过程，寻找相同的部分

不相交(disjoint)：一个实体至多属于一个底层实体集 重叠(overlapping)：一个实体可以同时属于同一个概化中的多个底层实体集 eg. 上层为人，下层为学生和老师。不相交意味着一个人不可以同时为老师和学生，重叠则意味着一个人可以同时为老师和学生。

概化的约束

定义：在特定概化上设置约束 内容：中文课本P168

1. 条件定义(conditional-defined)
2. 用户自定义(user-defined)

表示为关系模式

1. 概化的表示
    1. 重叠情况下(不相交情况下可以用) person(ID, name, age) student(ID, grade) teacher(ID, salary) 注：可在student, teacher的ID上设置外码
    2. 不相交情况下 student(ID, name, age, grade) teacher(ID, name, age, salary)
2. 聚集的表示 将聚集（联系集的属性）和其他实体集一样看待

第8章 关系数据库

数据依赖

关系模式中的各属性之间相互依赖、相互制约的联系称为数据依赖。

1. 函数依赖(function dependency)
    - 定义：满足函数依赖F:$\alpha \to \beta$：对实例中所有元组t_1和t_2，若$t_1[\alpha] = t_2[\alpha]$，则$t_1[\beta] = t_2[\beta]$
    - 分类
        1. 平凡(trivial)函数依赖：$\alpha \to \beta, \beta \subseteq \alpha$
        2. 非平凡函数依赖：$\alpha \to \beta, \beta \not\subset \alpha$
        3. 部分函数依赖$\alpha \xrightarrow{p} \beta$：满足函数依赖$\alpha \to \beta, \exists \alpha^{'} \subseteq \alpha, \alpha^{'} \to \beta$
        4. 完全函数依赖$\alpha \xrightarrow{f} \beta$：满足函数依赖$\alpha \to \beta$，不存在$\alpha^{'} \subseteq \alpha, \alpha^{'} \to \beta$ 推论：当决定因素（箭头左边）是单值属性的话，只能是完全函数依赖
        5. 传递函数依赖 γ完全传递依赖于α：$\alpha \to \beta, \beta \not\subset \alpha, \beta \to \gamma$
    - 函数依赖的特点：
        1. 当属性α和属性β的关系为1:1：$\alpha \leftrightarrow \beta$
        2. 当属性α和属性β的关系为m:1：$\alpha \to \beta$
        3. 当属性α和属性β的关系为m:n：不存在任何函数依赖
    - 函数依赖集F
        - F的闭包F^+：从F集合推导出的所有函数依赖的集合。

        - 方法一：

$$
\begin{aligned}
&F^+ = F \\
&\textbf{repeat} \\
&\quad \textbf{for each} \text{ functional dependency } f \text{ in } F^+ \\
&\quad\quad \text{apply reflexivity and augmentation rules on } f \\
&\quad\quad \text{add the resulting functional dependencies to } F^+ \\
&\quad \textbf{for each} \text{ pair of functional dependencies } f_1 \text{ and } f_2 \text{ in } F^+ \\
&\quad\quad \textbf{if } f_1 \text{ and } f_2 \text{ can be combined using transitivity} \\
&\quad\quad\quad \text{add the resulting functional dependency to } F^+ \\
&\textbf{until } F^+ \text{ does not change any further}
\end{aligned}
$$

        - 方法二：在属性集的闭包的作用
    - $F \iff G$：$F^+ = G^+$
  - Armstrong's axiom system
    1. reflexivity rule：$\beta \subseteq \alpha, \alpha \to \beta$
    2. augmentation rule：$\alpha \to \beta, \gamma\alpha \to \gamma\beta$
    3. transitivity rule: $\alpha \to \beta, \beta \to \gamma, \alpha \to \gamma$
    4. union rule: $\alpha \to \beta, \alpha \to \gamma; \alpha \to \beta\gamma$
    5. decomposition: $\alpha \to \beta\gamma; \alpha \to \beta, \alpha \to \gamma$
    6. peseudotransitivity rule: $\alpha \to \beta, \gamma\beta \to \delta; \alpha\gamma \to \delta$
2. 多值依赖(Multivalued Dependencies)：不考
3. 连接依赖：不考

符号表示说明

课本的表示方法：`r(R)`，r表示一个关系模式，R表示这个关系模式的属性集（特殊），通常用希腊字母(如α)表示属性集。元组(表的行)使用t表示，大写字母表示属性

PPT的表示方法：`R(U)`，R表示一个关系模式，U表示这个关系模式的属性集，通常用大写字母表示属性集。关系模式可以表示为R<U,F>，其中F为关系模式满足的函数依赖，元组(表的行)使用t表示

属性集的闭包

函数确定(functionally detemine)：如果函数$\alpha \to B$，那么称属性B被α函数确定 属性集的闭包：令α为一个属性集，在函数依赖集F下被α函数确定的属性的集合，记为α^+

属性集的闭包的算法：

- 方法一：计算F^+，找出箭头右边属性的并集
- 方法二：

```
result := α;
repeat
    for each functional dependency β → γ in F do
        begin
            if β ⊆ result then result := result ∪ γ;
        end
until (result does not change)
```

作用：

1. 判断α是否为超码
2. 若$\beta \subseteq \alpha^+$，则$\alpha \to \beta$成立，即$\alpha \to \beta \subseteq F^+$
3. 一种新的计算F^+，$\forall \gamma \subseteq R$，计算$\gamma^+$，将$\gamma \to S$加入到$F^+$中，其中S满足$\forall S \subseteq \gamma^+$

正则覆盖(canonical cover)

无关属性(extraneous attribute)

无关属性：去除函数依赖中的一个属性不改变该函数依赖集的闭包

定义：设F为函数依赖集，$\alpha \to \beta$

1. 如果A∈α，F逻辑蕴含(F - {α → β}) ∪ { (α - A) → β}，则A在α是无关的。例如：F上有函数依赖AB→C和A→C，那么B在AB→C的左半部是无关的.
2. 如果A∈β，(F - {α → β}) ∪ { α → (β - A)}逻辑蕴含F，则A在β是无关的。例如：F上有函数依赖AB→CD和A→C，那么C在AB→CD的右半部是无关的。

定义的备注：

- A逻辑蕴含(logically imply)B，表示A可以推出B
- 上面的-表示差集
- 定义的1中，F逻辑蕴含(F - {α → β}) ，这个条件肯定为真，只需要简要F逻辑蕴含{ (α - A) → β}

正则覆盖

正则覆盖（最小覆盖）：F的正则覆盖F_c是一个依赖集，F逻辑蕴含F_c，F_c逻辑蕴含F

特点：

- F_c 中的任何函数依赖都不含无关属性
- F_c 中函数依赖的左半部都是唯一的，也就是F_c中不存在两个依赖：$\alpha_1 \to \beta_1$ 和$\alpha_2 \to \beta_1$

正则覆盖的计算：

$F_c = F$
repeat
  Use the union rule to replace any dependencies in F_c of the form $\alpha_1 \to \beta_1$ and $\alpha_1 \to \beta_2$ with $\alpha_1 \to \beta_1 \beta_2$.
  Find a functional dependency $\alpha \to \beta$ in F_c with an extraneous attribute either in α or in β.
    /* Note: the test for extraneous attributes is done using F_c, not F */
  If an extraneous attribute is found, delete it from $\alpha \to \beta$ in F_c.
until (F_c does not change)

无损分解(lossless decomposition)

$R < U, F >$分解定义：$\rho = R_1 < U_1, F_1 >, R_n < U_n, F_n >, U = U_1 \bigcup ... \bigcup U_n, U_i \not\subset U_j, i \neq j$ 无损分解：$\Pi_{R_1}(r_1) \bowtie \Pi_{R_2}(r_2) = r$ 证明是无损分解：$\rho = R_1, R_2$，满足下面其中的一个即可

1. $R_1 \bigcap R_2 = (R_1 - R_2)$
2. $R_1 \bigcap R_2 = (R_2 - R_1)$

意义：保持无损分解意味着它不会丢失信息

保持函数依赖(Preserve dependency)

投影：Z是U的子集，函数依赖集合F在Z上的投影定义为
$\prod_Z(F) = \{X \to Y | X \to Y \in F^+ \vee XY \subseteq Z\}$ 保持函数依赖的分解：
$F^+ = (\bigcup_{i=1}^n \prod_{R_i}(F))^+ = (\bigcup_{i=1}^n F_i)^+$ 意义：保持了函数依赖，则它可以减轻或解决各种异常情况

范式

定义：范式是对关系的不同数据依赖程度的要求

主属性：候选码中的属性 非主属性：不包含在任何一个候选码中的属性

第一范式(first normal form)

定义：关系模式中所有属性的域都是原子的

意义：较细的原子粒度有助于标准化，施加约束，避免输入错误，从而提高数据质量

2NF

定义：若R∈1NF，且每个非主属性完全依赖于R的每一个候选关键字 ，则称R∈2NF 1NF → 2NF（规范化过程中通过一组投影运算消除部分依赖）例子：

```
已知关系R(A,B,C,D)，(A,B)为主码，即(A,B) → C,(A,B) → D，且A → D，则将R分解成为两个投影：
R1(A,D)，A为主码
R2(A,B,C)，(A,B)为主码，A为外码
```

3NF

定义：任何一个非主属性都不传递依赖于它的任何一个侯选关键字。也就是关系模式R< U , F >中，若不存在这样的码X，属性组Y及非主属性Z(Z⊄Y)，使得下式成立，X→Y , Y→Z , 而Y→X不成立。

2NF → 3NF的例子：

```
已知关系R(A,B,C)，A为主码(A → B，A → C)，且 B → C,则将R分解成为两个投影：
R1(B,C)，B为主码
R2(A,B)，A为主码，B为外码
```

将模式转化为3NF的保持函数依赖且无损的算法

let F_c be a canonical cover for F;
$i := 0$;
for each functional dependency $\alpha \rightarrow \beta$ in F_c
    $i := i + 1$;
    $R_i := \alpha\beta$;
if none of the schemas R_j, $j = 1, 2, \ldots, i$ contains a candidate key for R
  then
    $i := i + 1$;
    $R_i :=$ any candidate key for R;
/* Optionally, remove redundant relations */
repeat
    if any schema R_j is contained in another schema R_k
      then
        /* Delete R_j */
        $R_j := R_i$;
        $i := i - 1$;
until no more R_js can be deleted
return $(R_1, R_2, \ldots, R_i)$

例子：

设有关系模式R(SNO，SN，P，C，S，Z)
F={SNO→SN，SNO→P，SNO→C，SNO→S，SNO→Z，
{P,C,S}→Z，Z→P，Z →C}，试分解R为3NF。
解：
求F的最小覆盖Fc={SNO→{SN,P,C,S}，{P,C,S}→Z，Z→{P,C}}
根据上述算法，则分解为R1={SNO，SN,P,C,S}，R2={P,C,S,Z},R3={Z,P,C}，
∵R3是R2的子集，去掉R3
R1={SNO，SN,P,C,S}，F1={SNO→{SN,P,C,S}}
R2={P,C,S,Z}，F2={ {P,C,S}→Z，Z→{P,C} }
注意：R1,R2均属于3NF，但R1属于 BCNF，R2不属于BCNF ，若进一步将 R2 分解为 {P,C,Z} 和 {S,Z} ，则均为 BCNF ，但丢失了函数依赖 Z → {P,C}.

BCNF

定义：关系模式R< U , F >中，对于属性组X，Y，若X→Y且 Y ⊄X时，X必含有码。即：BC范式要求所有非平凡函数依赖都形如X→Y，其中X是一个超码。

特点：消除了主属性之间的相互依赖

BCNF的分解算法（一定无损，但不一定函数依赖）

result := {*R*};
done := false;
compute F^+;
while (**not** *done*) **do**
  if (there is a schema R_i in *result* that is not in BCNF)
    then begin
      let $\alpha \rightarrow \beta$ be a nontrivial functional dependency that holds on R_i such that $\alpha \rightarrow R_i$ is not in F^+, and $\alpha \cap \beta = \emptyset$;
      result := (*result* − R_i) ∪ (R_i − β) ∪ (α, β);
    end
  else *done* := true;

例子：

R=<U，F>，U={A，B，C，D，E}
F={A→B，B→C，(A,D)→E} ,求R的一个BCNF分解
解：1．候选码码是{AD}
2．检查A→B，由于A不是码，因此
U 1 ={A，B} ，F 1 ={A→B}
U 2 ={A，C，D，E}，F 2 ={A→C，(A,D)→E}
3．检查A→C，由于A不是码，因此
U 1 = {A，B}，F 1 ={A→B}
U 2 = {A，C}，F 2 ={A→C}
U 3 = {A，D，E}，F 3 = {(A,D)→E}

U=(SNO，TNO，CNO)
F={(SNO，CNO)→TNO，TNO→CNO}
解：不属于BCNF，分解为
U 1 =(SNO，TNO)，
U 2 =(TNO，CNO)，F 2 ={TNO→CNO}
丢失了函数依赖(SNO，CNO)→TNO，原来一个学生选修一门课程时，只能对应一个老师；在新的关系模式下现在一个学生选修一门课程时，可能会对应多个老师。关系分解为 BCNF ，不一定能保持函数依赖

第三部分 数据存储和查询

第11章 索引与散列

基本概念

查询处理方式：

- 文件扫描(file scaning)
- 索引(indices)

基本索引类型

- 顺序索引(ordered indices)：基于值的顺序排列
- 散列索引(hash indices)：基于将值平均分布到若干散列桶中

评价索引标准

- 访问类型(access type)
- 访问时间(access time)
- 插入时间(insertion time)
- 删除时间(deletion time)
- 空间开销(space overhead)

搜索码(search key)：在文件中用于查找的属性或属性集

顺序索引

分类

索引项（索引文件的一项）：搜索码+指向记录的指针

根据记录文件的顺序分类：

- 聚集索引(clustering index)（主索引，primary index）：包含记录的文件按照搜索码排序
- 非聚集索引(unclustering index)（辅助索引，secondary index）：与上面相对的索引

根据搜索码和索引项的对应情况来分类：

- 稠密索引(dense index)：每个搜索码值都有一个索引项
- 稀疏索引(sparse index)：只为某些搜索码建立索引项（稀疏索引只有在聚集索引的时候才有意义）

稠密索引的特点：

- 记录通常比索引项要大
- 可以快速查找索引（例如二分法）
- 索引可以常驻内存，使得搜索码值为K 的记录是否存在，不需要访问磁盘数据块

稀疏索引的特点：

- 节省了索引空间
- 对于“是否存在码值为 对于“是否存在码值为K的记录？”，需要访问磁盘数据块

稀疏索引如何查找：在索引文件中找到第一个小于或等于搜索目标的索引项

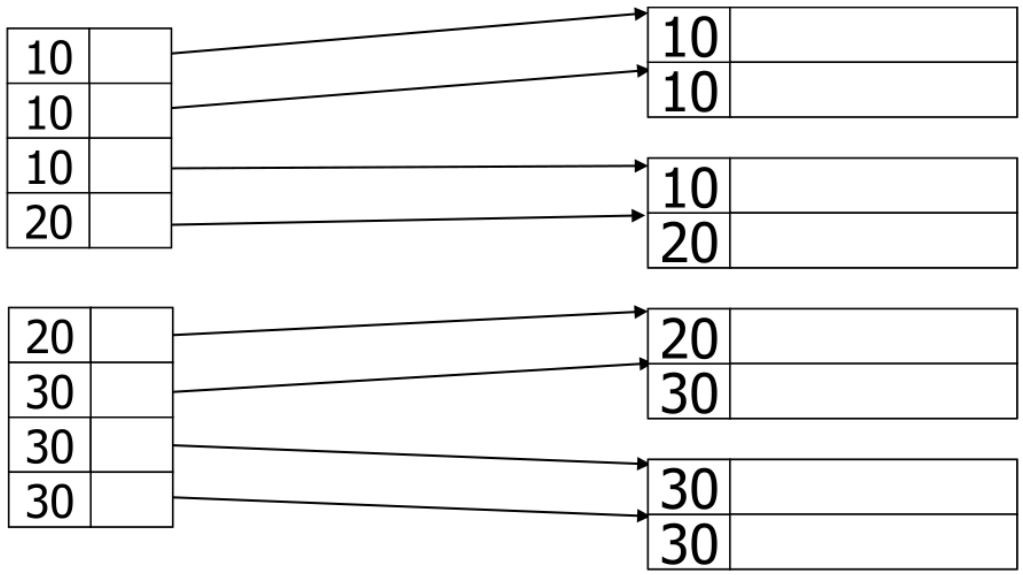
两者比较：

- 稠密索引能更快找到记录

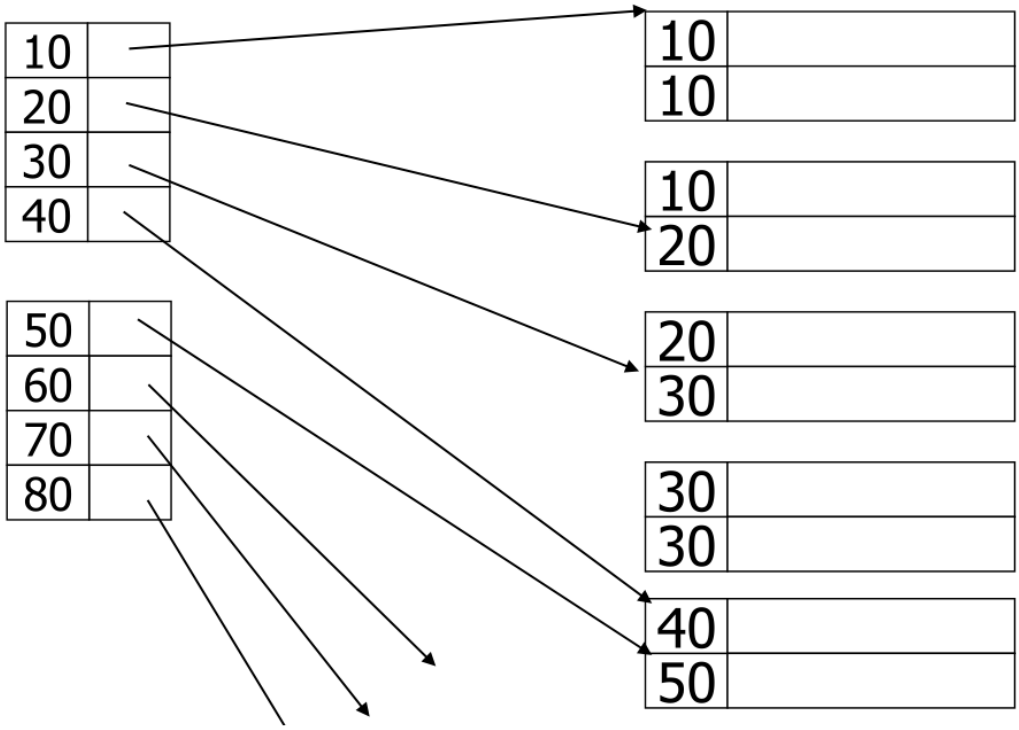
- 稀疏索引占用空间小，插入删除维护开销小

两者在有重复搜索码的处理情况

- 稠密索引
  - 为重复码值建立重复索引项

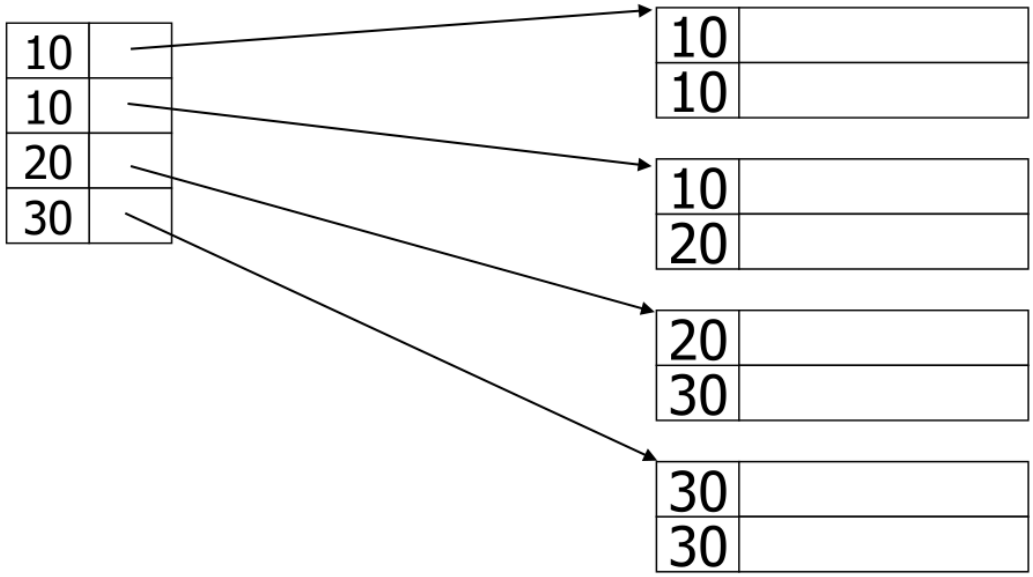


  - 每个不同的码值建立一个索引项

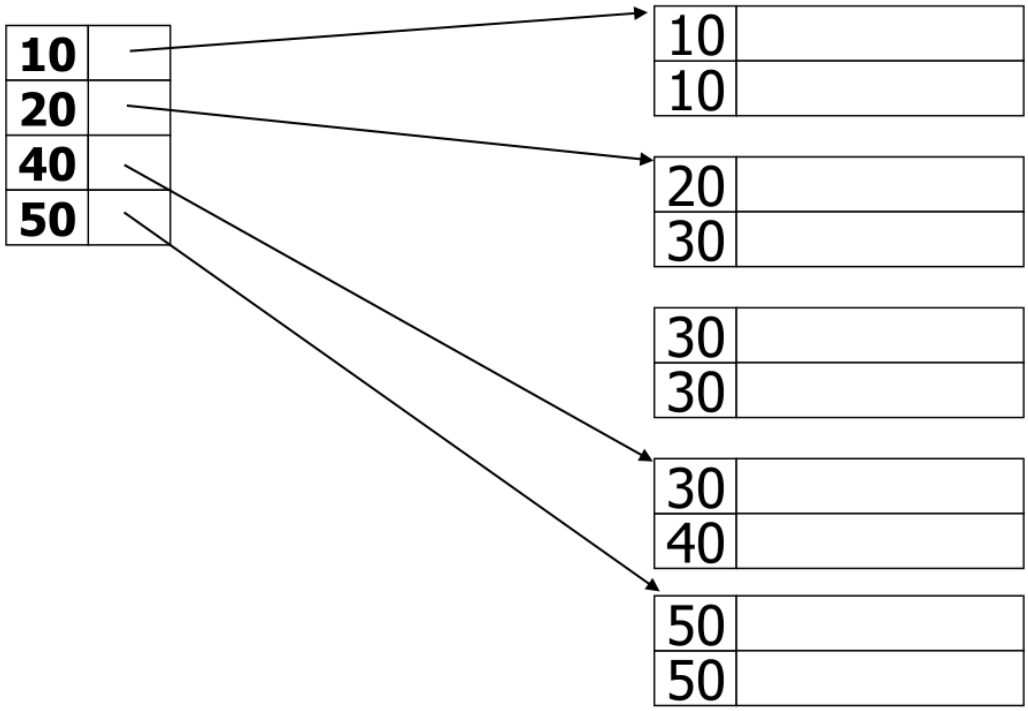


- 稀疏索引

- 为每个块的第一个记录建立索引项



- 为每个块的最小新码值建立索引项



多级索引

多级索引和树结构紧密相关 多级索引的最外层索引必定是稀疏索引，不可能是稠密索引，是为了减少索引文件的大小。

索引的更新

老师的ppt是给了几个例子，详情见中文课本P272

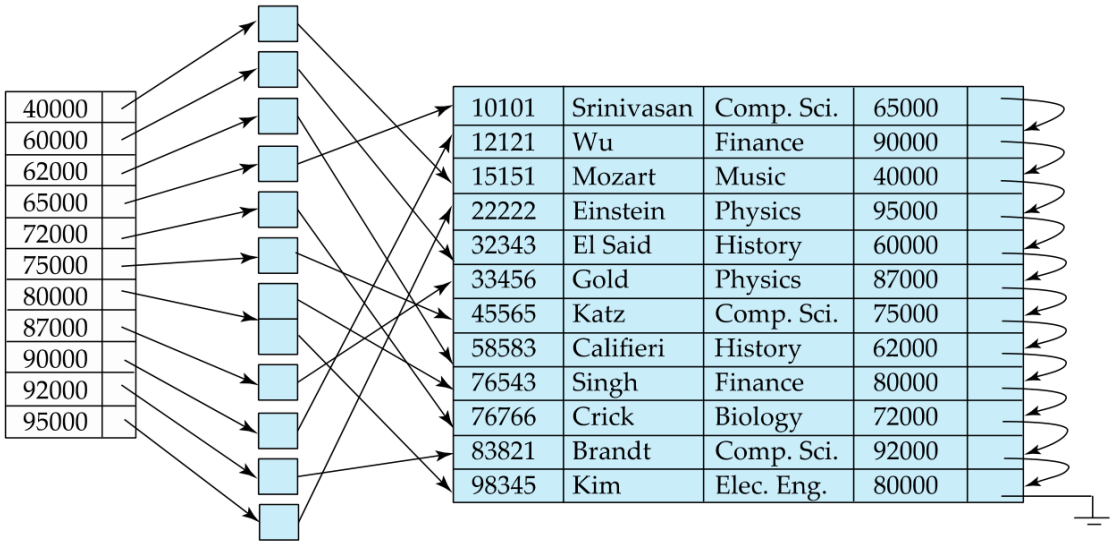
辅助索引

辅助索引必须是稠密索引，聚集索引可以是稀疏索引

索引项：

- 搜索码是候选码：搜索码+记录指针

- 搜索码不是候选码：搜索码+指针桶指针



B^+树

简介

- 一种树型的多级索引结构
- 树的层数与数据大小相关，通常树的层数与数据大小相关，通常为3层
- 所有结点格式相同：n -1个搜索码值，n个指针
- 平衡树结构(balanced tree): 所有叶结点位于同一层
- 适用于主索引，也可用于辅助索引

叶子节点

- 至少$\lceil \frac{n-1}{2} \rceil$ 个指针指向码值，至多$n-1$
- 1个指针向下一个相邻叶结点的指针

内部节点

- 第i个码值是第i+1个子树中的最小码值
- 至少$\lceil \frac{n}{2} \rceil$个指针指向子树，至多$n$
- 根结点至少2个指针，也就是说可以少于$\lceil \frac{n}{2} \rceil$

查找算法

```
Find record with search-key value V.
    C = root
    While C is not a leaf node
        Let i be least value s.t. V ≤ Ki.
        If no such exists
            set C = last non-null pointer in C
        Else if V = Ki
            Set C = Pi +1
        else set C = Pi
    Let i be least value s.t. Ki = V
    If there is such a value i
        follow pointer P i to the desired record.
    Else
        no record with search-key value k exists.
```

如果有K个记录，树的高度不会超过$\lceil log_{\lceil \frac{n}{2} \rceil}(K) \rceil$

B^+树操作重新看数据结构

效率

- 访问索引的I/O代价
  - 树高（B＋树不常驻内存）
  - 0（常驻内存）
- 树高通常不超过3层，因此索引I/O代价不超过3（总代价不超过4） 通常情况下，根节点常驻内存，因此索引I/O代价 代价不超过2 （总代价不超过3）

散列索引

静态散列

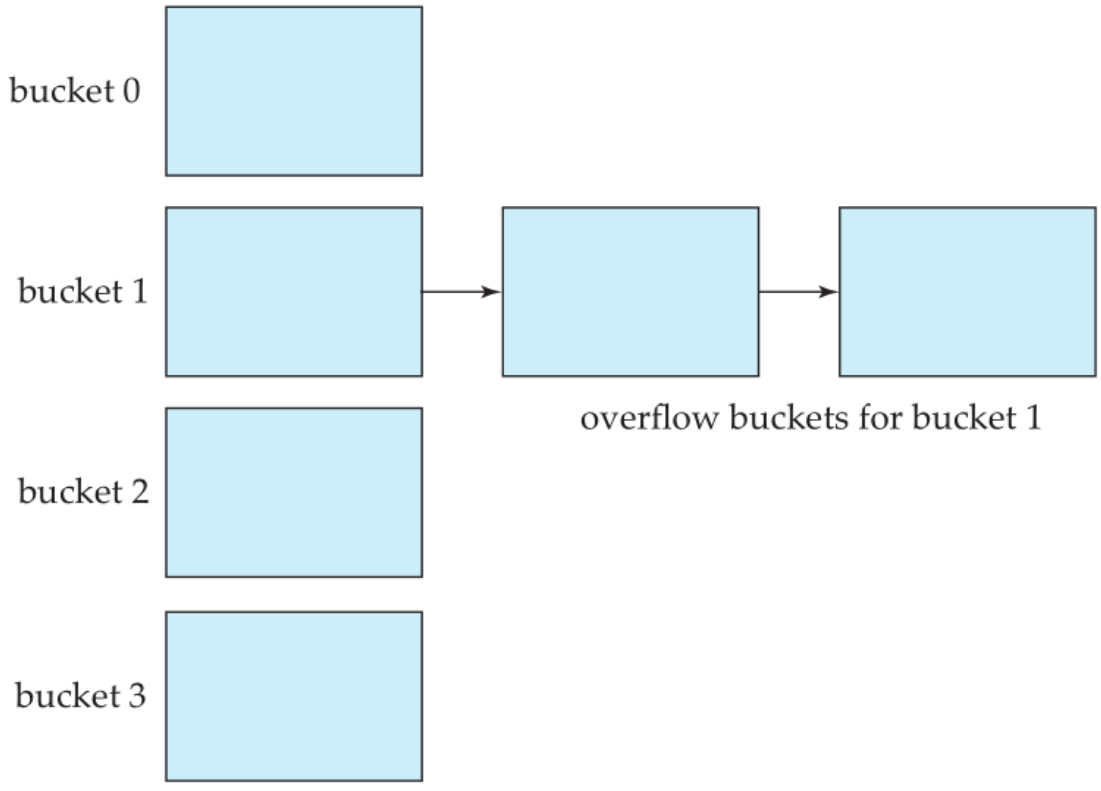
每个桶里可以有多个记录，桶与磁盘块的数量关系不一定是1:1，故有B个桶，记录的总块数为n，IO为$\lceil \frac{n}{B} \rceil$

查找

1. 计算h(K)
2. 根据h(K)定位桶
3. 查找桶中的块

插入

1. 计算插入记录的h(K)来定位桶
2. 若桶中有空间，则插入
3. 若没有，创建一个溢出桶并将记录置于溢出桶中（开散列(open index)的处理方法）



空间利用率 $= \frac{\text{实际码值数}}{\text{桶的总码值数}}$(50%-80% is good)

动态散列(dynatic hashing)

目的：解决数据库随时间增大而溢出桶增多的问题

分类：

- 可扩展散列表(Extensible Hash Tables)：成倍增加桶数目
- 线性散列表(Linear Hash Tables)：线性增加

可扩展散列表

散列值h(k)是一个b(足够大)位二进制序列，前i位表示桶的数目。 位表示桶的数目。i的值随数据文件的增长而增大

桶地址表(bucket address table)是存在在内存中

优点：当查找记录时，只需查找一个存储块。 缺点：桶增长速度快，可能会导致内存放不下整个桶数组，影响其他保存在主存中的数据，波动较大。

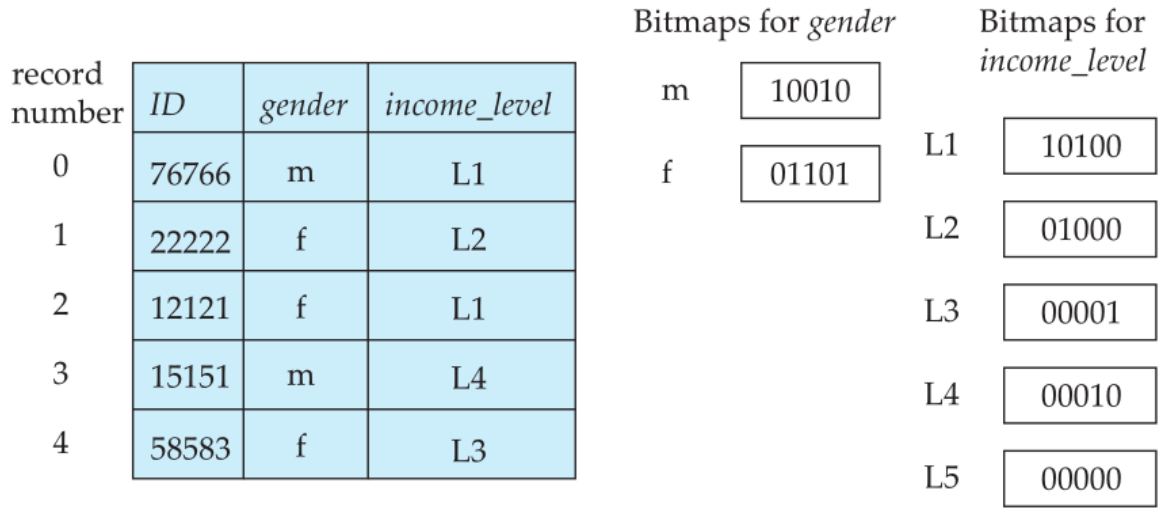
线性散列表

h(k)仍是二进制位序列，但使用右边(低)i位区分桶。桶数＝ n，h(k)的右i位＝ m，记录总数为r

- 若$m < n$,则记录位于第m个桶
- 若$n \leq m < 2^i$, 则记录位于第$m - 2^{i-1}$个桶
- n的选择：总是使n与当前记录总数r保持某个固定比例，如$r \leq 1.7n$，不满足条件时，桶是线性增加的

位图(bitmap)索引

一种为多码上的简单查询设计的特殊索引。 位图是一个有N 位的序列（每一位的值是0 或者1）， 第i位表示table的第i行信息



record number	ID	gender	income_level
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

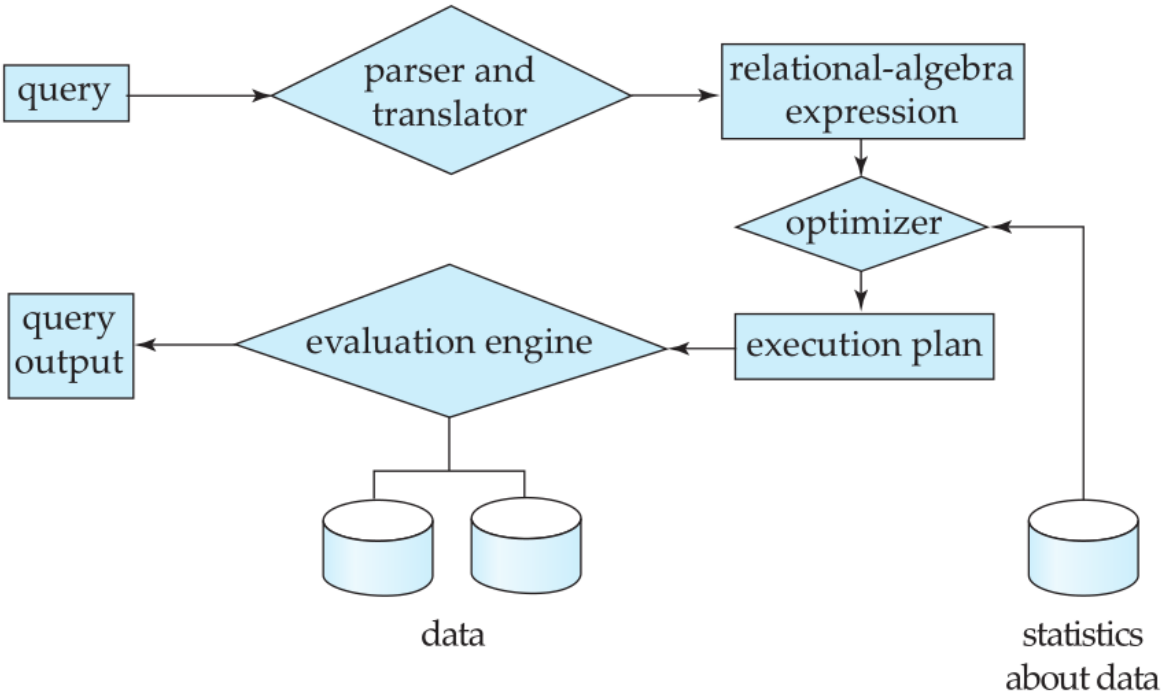
m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

在gender的m位图中，第一为0表示table第一行的gender属性不是m，若为1，则表示是m

第12章 查询处理



执行计划：给某些执行添加注释/说明，例如使用哪个索引进行查找，使用什么排序算法进行排序

查询

单字段：

- 没有索引：顺序扫描
- 主索引(B^+tree)：$h+\lceil\frac{s}{p}\rceil$
  - h : 树高
  - p：磁盘块的平均记录数
  - s：满足条件的记录数
- 辅助索引(B^+)：$(h-1)+\lceil\frac{k}{f}\rceil+s$
  - h：树高
  - f：磁盘块存储的叶子的个数
  - k：符合条件的叶子的个数
  - s：符合条件的记录数 注：一般来说，k < s，因为叶子存储的是指针桶的地址，一个指针桶含着多个记录的指针

多字段查询：

- 没索引：顺序扫描
- 一个字段有索引，一个字段没有索引：先查找满足有索引字段的记录，再去除那些没有满足没有索引的记录，代价取决于有索引的代价
- 都有索引：用各自的索引查找各自满足字段的记录指针，再将指针取交集。
  $(h_1-1)+\lceil\frac{k_1}{f_1}\rceil+(h_2-1)+\lceil\frac{k_2}{f_2}\rceil+s$

外排序：

符号表示：

- n：文件的磁盘块数
- M：内存的块数 注：假设磁盘块与内存块的大小一样。

过程：

1. 预排序(preliminary sorting)：形成顺串(run)

- 读写各一次
- 顺串数$N = \lceil \frac{n}{M} \rceil$
- $IO = 2n$

2. 归并(merging)
   - M-1路归并
   - 归并的次数x满足$(M-1)^x = N$，即$x = \log_{M-1} \lceil \frac{n}{M} \rceil$
   - 每次归并的IO=2n，故总$IO = 2n \cdot x$

- $IO = 2n(1 + \log_{M-1} \lceil \frac{n}{M} \rceil)$

连接（join）

二路连接：对两个文件进行连接。（我们只学习二路连接）

嵌套连接(nested loop)

- **设有关系*R*和*S*进行连接操作：$R \bowtie_{\theta} S$**

for each tuple t_R ***in*** *R* ***do begin***
  for each tuple t_S ***in*** *S* ***do begin***
    test pair (t_R,t_S) to see if they satisfy the join condition θ
      if they do, add $t_R \bullet t_S$ to the result.
  end
end

- ***R* is called the outer relation and *S* the inner relation of the join.**

块嵌套循环连接(block nested loop join, BNL)

已物理块为信息传递单位，面向block的连接

- 磁盘←→内存：以物理块（block）为信息传递的单位，面向block的连接称为块嵌套循环连接（BNL）

for each block B_R *of* R *do begin*
  for each block B_S *of S do begin*
    for each tuple t_R *in* B_R *do begin*
      for each tuple t_S *in* B_S *do begin*
        Check if (t_R,t_S) satisfy the join condition
          if they do, add $t_R \bullet t_S$ to the result.
      end
    end
  end
end

符号表示：T表示记录文件，n表示记录文件的快数，M表示内存的块数

当$n_1 < n_2$，读取T_1的$M-1$个块进入内存，剩下的一个块给T_2

IO次数：

- $M \geq n_1 + 1$：$n_1 + n_2$
- $M < n_1 + 1$
  - T_1：n_1
  - T_2：$n_2 * \frac{n_1}{M-1}$，其中$\frac{n_1}{M-1}$为外循环的次数
  - 总的$IO = n_1 + n_2 * \frac{n_1}{M-1}$

索引嵌套循环连接(indexed nested loop join)

- 一个没有索引，一个使用B^+索引
- 一个没有索引，一个使用哈希索引

排序-归并连接(sort-merge join)

过程：

- 使用外排序对两个文件进行排序
- 使用类似外排序的归并排序对两个有序文件进行连接，此步骤的$IO = n_1 + n_2$

总IO：$2n_1(1 + \log_{M-1} \lceil \frac{n_1}{M} \rceil) + 2n_2(1 + \log_{M-1} \lceil \frac{n_2}{M} \rceil) + n_1 + n_2$

散列连接(hash join)

满足最小桶的磁盘块数 + 1 ≤ 内存的快数 在内存中进行的桶连接的方法是上面的块嵌套连接

见ppt

查询处理

集合的交、并、差集都使用hash实现

处理多个计算：

- 物化(meterialized)（实体化）计算(evaluation)：每次执行操作，就在磁盘上建立临时文件来保存操作的临时结果
- 流水线计算(pipelined evaluation)：上一个操作的结果元组传到下一个操作的输入以减少临时文件数。

第13章 查询优化

分类：

- 代数优化：优化方法仅涉及查询语句本身
- 物理优化：依赖于存取路径的优化

代数优化

原则：尽量减少查询过程中的中间结果大小

关系表达式的等价(equvialence)：在一个相同的数据库实例中输出相同的元组集。

等价规则(equivalance rules)：中文课本P331

基于代价的优化器(cost-based optimizer)：从给定查询等价的所有查询执行计划空间中进行搜索，并选择估计代价(estimated)最小的一个

启发式(heuristics)优化

引入：基于代价的优化器本身优化执行计划也有一定的代价，需要降低这些代价 目的：减少优化代价 原则：

- 尽早执行选择运算(Perform selection operations as early as possible.)
- 尽早执行投影运算(Perform projections early.)

第14章 事务

事务是访问并可能更新各种数据项的一个程序执行单元。

特性（ACID）：

- 原子性(atomicity)：要么全部执行，要不全部不执行。
- 一致性(consistence)
- 隔离性(isolation)：执行的时候不能被其他干扰
- 持久性(durability)：影响是永久的，即使系统奔溃，重启继续执行

事务的状态：

- 活动的(Active)：正在执行
- 部分提交的(Partially committed)：最后一条语句被执行后
- 失败的(failed)：发现正常的执行继续后
- 中止的(aborted)：事务回滚到执行前
- 提交的(committed)：成功完成后

原子性和永久性的实现：在副本操作，db_point指向当前的数据库，事务成功则更新db_point

可串行化(serializable)：P363

一些重要的概念

- 调度(schedule)：指令在系统中执行的时间顺序
- 可串行化调度(serializable schedule)：P363，在并发的过程中，通过保证执行的任何调度效果与没有并发前一样，这种并发的调度称为并发调度，没有并发前的调度是可串行化调度的，即可以并发。
- 冲突等价(conflict equivalant)：P364，如果调度S可以通过一系列非冲突指令交换转换成S'，那么称S和S'冲突等价
- 冲突可串行化(conflict serializable)：P364，如果调度S与一个串行调度冲突等价，则称调度S是冲突串行化的。
- 优先图(precedence graph)：P364
- 视图可串行化(view serializable)：P385

冲突可串行化一定属于视图可串行化，视图可串行化属于可串行化的一种。

可恢复调度(recoverable schedule)：如果T_i读取了之前由T_j所写的数据，T_i必须比T_j先提交(commit)。P366

级联回滚(cascading rollback)：一个事务的回滚导致其他事务跟着回滚。

无级联回滚调度(cascadeless schedule)：没有级联回滚的调度，也是调度的目标

第15章 并发控制(concurrency control)

并发(concurrency)：一个CPU

并行(parallel)：多个CPU

Lock

分类：

- 共享型(shared-mode, S) ：只读
- 排他型(exclusive-mode, X)：读写

锁的兼容性(compatibility)：不同锁是否可以同时加在同一个数据上。只有S和S是兼容的。

可能引发的问题

- 死锁(deadlock)
- 饿死(starvation)

DeadLock

两种处理方法

- 事前处理：死锁预防(deadlock prevention)
  - 通过对加锁请求进行排序 或 同时获得所有的锁（即要不全给，要不全不给）来保证不会发生循环等待
  - 使用抢占(preempt)和事务回滚
    - wait-die：nonpreempted，新的等待老的，老的抢新的，P380
    - wound-wait：preempted，新的抢老的，老的等新的，P380
    - lock timeout：P380
- 事后处理：死锁检查(deadlock detection)和死锁恢复(deadlock recovery) P380
  - 死锁检测：是否存在cycle
  - 死锁恢复
    - 选择牺牲者
    - 回滚

two -phase locking protocol

两阶段封锁协议：一种关于lock的授予方法，P376

- 增长阶段(grow phase)：事务可以获得锁，但不能释放锁
- 缩减阶段(shrinking phase)：事务可以释放锁，但不能获得锁。

特点：

- 保证冲突可串行化：记调度事务获得其最后加锁的位置（增加阶段结束点）为事务的封锁点(lock time)，多个事务可以根据它们的封锁点进行排序
- 不能保证不会发生死锁

变体：

- 严格两阶段的封锁协议(strict two-phase locking protocol)：保证不级联回滚
- 强两阶段封锁阶段(rigorous two-phase locking protocol)

锁转换(Lock Conversions)

- upgrade：S -> X，只能在增长阶段
- downgrade：X -> S，只能在缩减阶段

第16章 恢复系统

事务对数据库的修改类型：

- 延迟修改(deferred-modification)

- 立即修改(immediate-modificaiton)

日志记录(log record)

中文课本P406

特点：幂等(idempoent)，即无论所记录的更新执行一次还是多次，对应的数据项都会是同样的值。

每次事务执行写之前，必须在数据库修改前建立该次写操作的日志并将它加到日志中。

恢复算法（支持立即修改）

- undo
- redo

为什么对commit的事务进行redo：这一轻微的冗余简化了恢复算法