

# 数据库 (Database Systems)

AilanXierAilanXierAilanXierAilanXierAilanXierAilanXierAilanXierAilanXierAilanXierAilanXier.

数据库课程很难，知识琐碎，涉及科目广，考试开发性强，考巨多理解题，PPT 讲的不明所以，谜语人化严重，部分老师只会念 PPT，建议有条件旁听张晓峰老师的课，体验会好上不少，实在听不懂只能去看 MOOC 的战老师了，但是估计效果也不是很好（PPT 朗诵大师<sup>2</sup>）。

因为从 PPT 获取的信息很有限（或者说信息非常劣质），笔者只能以它的基础上（复制），在有限时间和能力上试图总结自己对数据库的理解（对 PPT 整理并批注），可能存在很多错误（主观上的，如带有「笔者认为」的地方，或笔误的），需要自己加以判断。因为不考 SQL 语句和没时间整理，没有总结 SQL 语句和关系代数的部分，需要后人填补。务必对关键概念有清楚理解，考试难度超乎想象，及格都难，开放题居多（淘宝，抖音策略）。

部分地方参考了 [『数据库系统概念（原书第六版）』](#)，希望有所帮助，这里放上我加了 OCR 和书签后书籍的阿里云盘链接。[.md](#)  
建议使用 Typora 查看，自定义主题为 Purple，位于文件夹的 purple.css。

感谢张晓峰老师, [diri233](#), [Hwcoder](#), [xyfJASON](#) 的指正。

对于考试有考到的地方有  标记

## 1.数据库基本概念

## 1.1.数据抽象

通过抽象来对用户屏蔽复杂性，以简化用户与系统的交互。

- #### • 物理层（或内部层）：

最低层次的抽象，描述数据实际上是**怎样存储的**和复杂的底层数据结构（存储路径、存储方式、索引方式）。

- 逻辑层（或概念层）：

比物理层层次稍高的抽象，描述数据库中存储什么数据及这些数据间关系。

- #### • 视图层（或外部层）：

最高层次的抽象，只描述整个数据库的某个部分。用于并不需要关心所有的信息，而只需要访问数据库的一部分的用户。同一数据库有多个视图。

- 实例：

特定时刻存储在数据库中的信息的集合称作数据库的一个实例。

- #### • 模式:

数据库的总体设计称作数据库模式（schema），是对数据库中数据所进行的一种结构性的描述。

数据库模式对应于程序设计语言中的变量声明（以及与之关联的类型的定义）。

每个变量在特定的时刻会有特定的值，程序中变量在某一时刻的值对应于数据库模式的一个实例。

- 在不同抽象层次描述数据库，就可定义出物理模式，逻辑模式和视图模式。

相同模式有不同名称：

视图英文		三级模式两层映像结构中的名字			
External Schema	外模式	局部模式	视图模式	用户模式	子模式
(Conceptual) Schema	概念模式	全局模式	逻辑模式	也可简称「模式」	
Internal Schema	内模式	物理模式	存储模式		

- #### • 两层映像:

- E-C Mapping (External Schema-Conceptual Schema Mapping) :

将外模式映射为概念模式，从而支持实现数据概念视图向外部视图的转换，便于用户观察和使用。

逻辑数据独立性：

当概念模式变化时，可以不改变外部模式（只需改变 E-C Mapping），从而无需改变应用程序。

◦ C-I Mapping (Conceptual Schema-Internal Schema Mapping) :

将概念模式映射为内模式，从而支持实现数据概念视图向内部视图的转换，便于计算机进行存储和处理。

物理数据独立性：

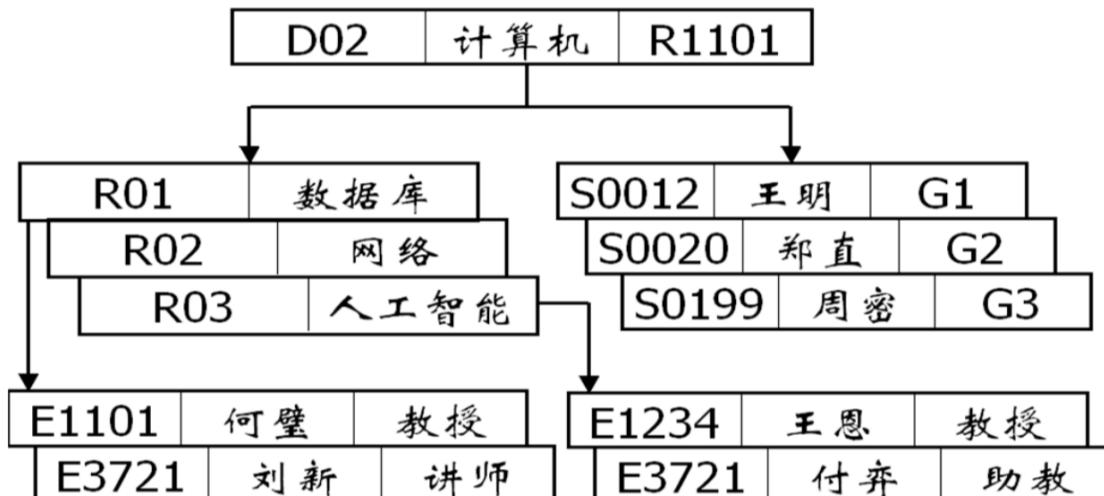
当内部模式（物理模式）变化时，可以不改变概念模式（只需改变 C-I Mapping），从而不改变外部模式。

## 1.2. 数据模型

数据库结构的基础是数据模型。

• PPT 中的分类（往年考过三个模型的优缺点）：

- 层次模型按树的形式组织数据：



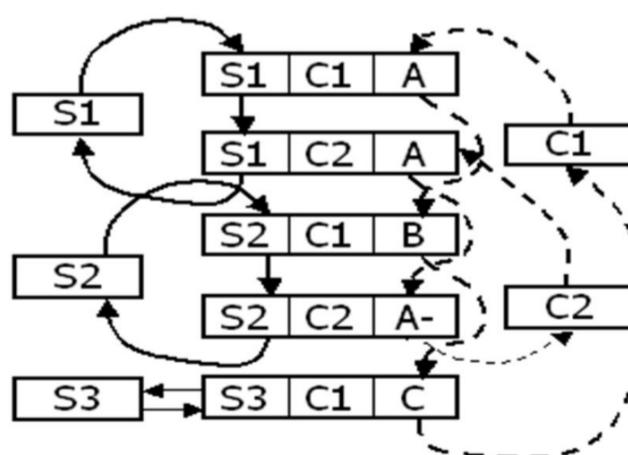
▪ 优点：

- 数据结构简单清晰。
- 查询效率高。记录之间的联系用有向边表示，这种联系在 DBMS 中通常使用指针实现。
- 性能优于关系数据库，不低于网状数据库。
- 提供了良好的完整性支持。

▪ 缺点：

- 对于非层次的联系（如多对多的联系）不适用。
- 一个子节点可能有多个父节点导致冗余，操作复杂。
- 必须通过父节点才能查找子节点。

◦ 网状模型按图的形式组织数据：



▪ 优点：

能够更为直接地表示现实世界，具有良好的性能，存取效率高。

▪ 缺点：

- 结构复杂。
- 网状模型的数据定义语言和数据管理语言复杂。
- 应用程序编写困难。

层次结构和网状结构共有的缺点：

- 数据之间的关联关系由复杂的指针系统来维系，结构描述复杂
- 数据检索操作依赖于由指针系统指示的路径
- 不能有效支持记录集合的操作

- 关系模型按表的形式组织数据：

优点：

- 数据检索操作不依赖于路径信息或过程信息，支持非过程化的数据操作
- 有效支持记录集合的操作
- 较为完善的理论基础

- 书上的分类：

- 关系模型：

用表的集合来表示数据和数据间的联系。

- 实体 - 联系 (E-R, entity-relationship) 模型：

- 基于对现实世界的认识。
- 现实世界由一组称作实体的基本对象以及这些对象间的联系构成。
- 实体是现实世界中可区别于其他对象的一件「事情」或一个「物体」

关系模型是数据模型，而 E-R 模型是概念模型。

- 基于对象的数据模型（应该不怎么学）

### 1.3.数据库系统和语言

- 数据库系统：

- 数据库 (DB, Database)
- 数据库管理系统 (DBMS, Database Management System)
- 数据库应用 (DBAP, DataBaseApplication)
- 数据库管理员 (DBA, DataBaseAdministrator)

- 数据库语言 (DBMS 提供的) :

- 数据定义语言 (DDL, Data Definition Language) :

定义表名，表标题，列名及其结构形式。

新数据插入时（更新数据库时）要检查完整性约束，防止不符合规范的数据进入数据库：

- 域约束：

每个属性都必须对应于一个所有可能的取值构成的域（例如，整数型、字符型、日期/时间型）。域约束是完整性约束的最基本形式。

- 参照（引用）完整性：

一个关系中给定属性集上的取值也在另一关系的某一属性集的取值中出现。

- 断言：

一个断言就是数据库需要始终满足的某一条件（域约束和参照完整性约束是断言的特殊形式），例如：

「每一学期每一个系必须至少开设 5 门课程」只能表达成一个断言。如果断言有效，则以后只有不破坏断言的数据库更新才被允许。

- 数据操纵语言 (DML, Data Manipulation Language) :

对数据库进行增、删、改、查等操作。

- 数据控制语言 (DCL, Data Control Language) :

对不同操作和用户的约束。

## 2.关系模型

## 2.1. 定义

- 表的列，也叫字段/属性/数据项，包括列名和列值。
- 表的行，也叫元组（ $n$  元组就是一个有  $n$  个值的元组）/记录。
- 域：  
对于关系的每个属性，都存在一个允许取值的集合（即域），这组值具有相同的数据类型。

集合中元素的个数称为域的基数。

- 域的笛卡尔积：

一组域  $D_1, D_2, \dots, D_n$  的笛卡尔积为： $D_1 \times D_2 \times \dots \times D_n = \{(d_1, d_2, \dots, d_n) | d_i \in D_i, i = 1, \dots, n\}$ ，笛卡尔积每个元素  $(d_1, d_2, \dots, d_n)$  称作一个  $n$ -元组。

- 关系：

一组域  $D_1, D_2, \dots, D_n$  的笛卡尔积的子集。

- 关系模式或表标题：

用  $R(A_1 : D_1, A_2 : D_2, \dots, A_n : D_n)$  表示，可简记为  $R(A_1, A_2, \dots, A_n)$  来描述关系：

- $R$  是关系的名字。
- $A_i$  是属性。
- $D_i$  是属性所对应的域（属性的类型、长度）。
- $n$  是关系的度或目，即有多少列。
- 关系模式中  $A_i (i = 1, \dots, n)$  必须是不同的，而  $D_i (i = 1, \dots, n)$  是可以相同的。

## 2.2. 关系的特性

- 列是同质：

每一列中的列值来自同一域，是同一类型的数据。

- 不同的列可来自同一个域

- 列位置互换性：

区分哪一列是靠列名，与列的顺序无关。

- 行位置互换性：

区分哪一行是靠某一或某几列的值（关键字/键字/码字）。

- 任意两个元组不能完全相同

- 属性不可再分特性（关系第一范式）

## 2.3. 码（键）

- 超码（superkey）：

超码是一个或多个属性的集合，这些属性的组合可以使我们在一个关系中唯一地标识一个元组。

两个不同元组的超码属性的取值不会完全相同。

- 候选码：

如果一个超码的任意真子集都不能成为超码，这样的最小超码称为候选码。

- 主码：

当有多个候选码时，可以选定一个作为主码。

- 主属性与非主属性：

包含在任何一个候选码中的属性被称作主属性，其他属性被称作非主属性。

如果关系的所有属性组是这个关系的候选码，称为全码。

- 外码：

关系  $R$  中的一个属性组，它不是  $R$  的候选码，但它与另一个关系候选码（主码）相对应，则称这个属性组为  $R$  的外码。

## 2.4. 关系模型中的完整性

- 实体完整性：

关系的主码中的属性值不能为**空值**。

- 参照完整性：

如果关系  $R_1$  的外码  $a$  与关系  $R_2$  的主码  $a$  相对应，则  $R_1$  中的每一个元组的  $a$  值或者等于  $R_2$  中某个元组的  $a$  值，或者为**空值**。

- 用户自定义完整性：

用户针对具体的应用环境定义的完整性约束条件。

这一章缺少了很关键的关系代数。

## 3. 数据库设计

### 3.1. 实体-联系 (E-R) 模型

- E-R 模型的基本观点：

世界是由一组称作**实体**的基本对象和这些对象之间的**联系**构成的。

- 实体：

**实体**是现实世界中可区别于所有其他对象的一个「事物」或「对象」。每个实体有一组性质（属性），其中一些性质的值可以唯一地标识一个实体。

- 实体集：

**实体集**是相同类型即具有**相同性质**（或属性）的一个实体集合。

注意：以上是书中的定义，PPT 中的「实体」=书中的「实体集」，PPT 的「实例」=书中的「实体」。

以下采用书本的定义描述。

- 联系集：

- **联系**是指**多个实体间**的相互关联。

- **联系集**是相同类型联系的集合。正规地说，联系集是  $n \geq 2$  个（可能相同的）实体集上的数学关系。如果  $E_1, E_2, \dots, E_n$  为实体集，那么联系集  $R$  是

$$\{(e_1, e_2, \dots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$

的一个**子集**（相当于实体集笛卡尔积的一个子集），而  $(e_1, e_2, \dots, e_n)$  是一个**联系**， $e_i$  是  $E_i$  的属性集合的子集。

实体集之间的联系称为**参与**。实体集  $E_1, E_2, \dots, E_n$  参与联系集  $R$ 。

举例：实体集 *Student*, *Course* 参与联系集 *SC*。

- 参与联系集的**实体集的数目**称为联系集的**度**。

- 实体在联系中扮演的功能称为**实体的角色**。

- 属性：

- 每个属性都有一个可取值的集合，称为该属性的**域**或者**值集**。

- 属性的分类：

- 简单（单一）属性：

- 它们不能划分为更小的部分。

- 复合属性：

- 将属性再划分为更小的部分（即其他属性）。

### 地址细分（省市）

- 单值属性：

对一个特定实体，一个属性都只有单独的一个值。

- 多值属性：

对某个特定实体而言，一个属性可能对应于一组值（即一个属性可以取多个值）。

### 一个人有多个电话号码

- 联系集的类型：

- 一对一：

$A$  中的一个实体至多与  $B$  中的一个实体相关联，并且  $B$  中的一个实体也至多与  $A$  中的一个实体相关联。

- 一对多：

$A$  中的一个实体可以与  $B$  中的任意数目（零个或多个）实体相关联，而  $B$  中的一个实体至多与  $A$  中的一个实体相关联。

- 多对一：

$A$  中的一个实体至多与  $B$  中的一个实体相关联，而  $B$  中的一个实体可以与  $A$  中任意数目（零个或多个）实体相关联。

- 多对多：

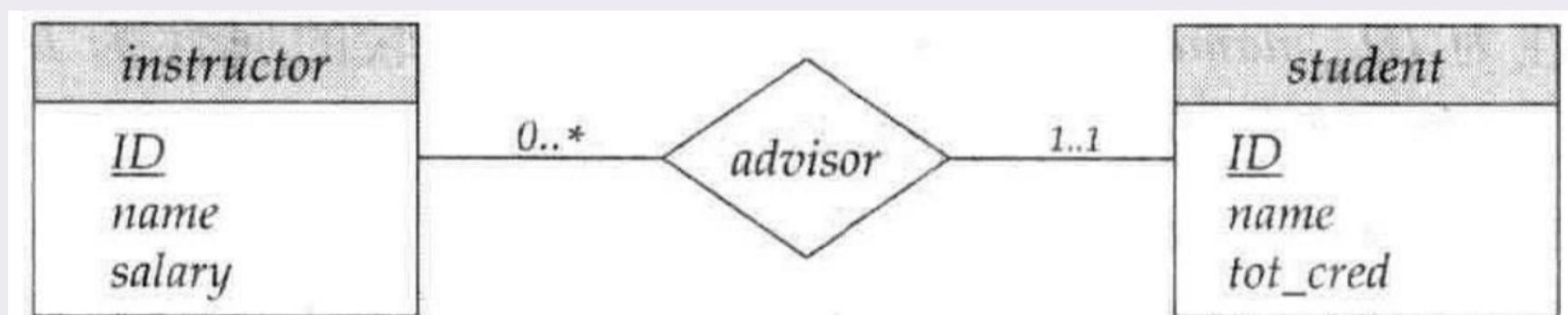
$A$  中的一个实体可以与  $B$  中任意数目（零个或多个）实体相关联，而且  $B$  中的一个实体也可以与  $A$  中任意数目（零个或多个）实体相关联。

- 映射基数（联系的基数\）：

实体之间的联系的数量，即一个实体通过一个联系能与另一实体集相关联的实体的数目。

- 实体集和二元联系集之间的一条边，可以有一个关联的最大和最小的映射基数。

例如：



一个 *instructor* 可以对应 0 个或多个 *student*，但一个 *student* 有且仅有一个 *instructor*，注意 *advisor* 是从 *instructor* 到 *student* 的一对多联系。

笔者注：

PPT 上的基数很多地方是反了，应该是自己的基数写在自己旁边（不知道祖传错误是否已被更正）

- 参与约束：

- 完全参与联系：

如果实体集  $E$  中的每个实体都参与到联系集  $R$  的至少一个联系中，此时最小基数为 1。

- 部分参与联系：

如果  $E$  中只有部分实体参与到  $R$  的联系中，此时最小基数为 0。

- 弱实体集和强实体集：

- 没有足够的属性以形成主码的实体集称作弱实体集（从属实体集）。

- 有主码的实体集称作强实体集（独立实体集）。

## 3.2. 数据库设计

- 需求分析：

形成数据库设计的「源」清单和「属性」清单。

- 概念数据库设计:

- 用统一的表达方法, 如 E-R 模型给出描述:

- 各种实体的发现、划分和定义
    - 各种实体属性的发现、分析和定义
    - 各种实体联系的发现、分析和定义
    - 外部视图(模式)和概念视图(模式)的定义

- 消除冲突:

- 属性冲突:

- 属性域的冲突(属性的类型、取值范围不同)

- 属性取值单位冲突

- 结构冲突:

- 同一对象在不同应用中的抽象不同(职工在某应用中是实体,在另一应用中则抽象为属性)

- 同一实体在不同 E-R 图中属性组成不同

- 实体之间的联系在不同 E-R 图中呈现不同的类型

- 命名冲突(同名异义, 异名同义)

- 逻辑数据库设计:

- E-R 图转换为关系模式:

- 复合属性转化:

- 拆成多列单一属性。

- 或者作为一个新的关系, 每个分量为新关系的属性。

- 多值属性转化:

- 将多值属性与所在实体的关键词一起组成一个新的关系。

- 也可以像复合属性一样拆成多列。

- 一对一联系:

- 若联系双方均部分参与, 则将联系定义为一个新的关系, 属性为参与双方的关键词(主码)属性。

- 若联系一方全部参与, 则将另一方关键词作为全部参与方的属性(即多一个属性), 不需要联系集(感觉是 PPT 说反了)。

- 一对多联系:

- 将单方(如教师和学生关系中的教师)参与实体的关键词, 作为多方(学生)参与实体对应关系的属性。

- 多对多联系:

- 将联系定义为新的关系, 属性为参与双方实体的关键词。

- 允余(没懂):

- 受控冗余:

- 有时为了效率和便利, 会特意设计冗余数据。

- 非受控冗余:

- 存在传递函数依赖。

### 3.3. 函数依赖

- 函数依赖定义:

- 设  $R$  是属性集合  $U = \{A_1, A_2, \dots, A_n\}$  上的一个关系模式,  $X, Y$  是属性集  $U$  上的两个子集。

- 若对  $R$  的任意关系  $r$ ,  $r$  中不可能有两个元组满足在  $X$  中的属性值相等而在  $Y$  中的属性值不等, 则称「 $X$  函数决定  $Y$ 」或「 $Y$  函数依赖于  $X$ 」, 记作  $X \rightarrow Y$ , 称  $X$  为决定因素,  $Y$  为依赖因素。

关系模式是在属性集合上的结构化描述, 关系是表状结构的。

函数依赖, 顾名思义,  $X$  映射到  $Y$ , 同一个  $x$  不会映射到两个  $y$ 。

- 函数依赖的特性:

- 如果  $Y \subset X$ , 则有  $X \rightarrow Y$ , 这是平凡的函数依赖。
  - 对  $X \rightarrow Y$ , 但  $Y \not\subset X$ , 则称  $X \rightarrow Y$  为非平凡的函数依赖。
  - 若  $X \rightarrow Y, Y \rightarrow X$ , 则记作  $X \leftrightarrow Y$ 。
  - 若  $Y$  不函数依赖于  $X$ , 则记作  $X \not\rightarrow Y$ 。

- $X \rightarrow Y$ , 若是基于模式  $R$  的, 则对任意的关系  $r$  成立; 若仅基于具体关系  $r$  的, 则可能只对该关系  $r$  成立。
- 如一关系  $r$  的某属性集  $X$ ,  $r$  中没有  $X$  上相等的两个元组, 则  $X \rightarrow Y$  恒成立。

• 完全函数依赖和部分函数依赖:

在  $R$  中, 若  $X \rightarrow Y$  并且对于  $X$  的任何真子集  $X'$  都有  $X' \not\rightarrow Y$ , 则称  $Y$  完全 (full) 函数依赖于  $X$ , 记为  $X \xrightarrow{f} Y$ , 否则称  $Y$  部分 (partial) 函数依赖于  $X$ , 记为  $X \xrightarrow{p} Y$ 。

$R$  是属性集合  $U$  上的一个关系模式, 若  $X \subset U$  且  $X \xrightarrow{f} U$ , 则称  $X$  为  $R$  的候选键 (候选码)。

• 传递函数依赖:

在  $R$  中, 若  $X \rightarrow Y, Y \rightarrow Z$ , 其中  $Y \not\subset X, Z \not\subset Y, Z \not\subset X, Y \not\rightarrow X$  (说明都是非平凡依赖), 则称  $Z$  传递函数依赖于  $X$ 。

传递依赖的判定很严格, 仅满足  $X \rightarrow Y, Y \rightarrow Z$  是不够的, 要特别注意。

• 逻辑蕴涵:

设  $F$  是关系模式  $R$  的一个函数依赖集合,  $X, Y$  是  $R$  的属性子集, 若从  $F$  的函数依赖能够推导出  $X \rightarrow Y$ , 则称  $F$  逻辑蕴涵  $X \rightarrow Y$ , 记作  $F \models X \rightarrow Y$ 。

• 闭包:

令  $F$  为一个函数依赖集,  $F$  的闭包是被  $F$  逻辑蕴涵的所有函数依赖的集合, 记作  $F^+$ 。若  $F^+ = F$ , 则  $F$  是一个全函数依赖族 (函数依赖完备集)。

在下面的规则中去寻找逻辑蕴涵的函数依赖, 通过反复应用这些规则, 可以找到给定  $F$  的全部  $F^+$ 。

用希腊字母  $(\alpha, \beta, \gamma, \dots)$  表示属性集, 用  $\alpha\beta$  表示  $\alpha \cup \beta$ 。

这组规则称为 Armstrong 公理 (据说不怎么会考) :

◦ 自反律:

若  $\alpha$  为一属性集且  $\beta \subseteq \alpha$ , 则  $\alpha \rightarrow \beta$  成立。

◦ 增补律:

若  $\alpha \rightarrow \beta$  成立且  $\gamma$  为一属性集, 则  $\gamma\alpha \rightarrow \gamma\beta$  和  $\gamma\alpha \rightarrow \beta$  成立。

◦ 传递律:

$\alpha \rightarrow \beta$  和  $\beta \rightarrow \gamma$  成立, 则  $\alpha \rightarrow \gamma$  成立。

公理可导出的比较好用的引理:

◦ 合并律:

若  $\alpha \rightarrow \beta$  和  $\alpha \rightarrow \gamma$  成立, 则  $\alpha \rightarrow \beta\gamma$  成立。

◦ 分解律:

若  $\alpha \rightarrow \beta\gamma$  成立, 则  $\alpha \rightarrow \beta$  和  $\alpha \rightarrow \gamma$  成立。

◦ 伪传递律:

若  $\alpha \rightarrow \beta$  和  $\gamma\beta \rightarrow \delta$  成立, 则  $\alpha\gamma \rightarrow \delta$  成立 (由  $\alpha \rightarrow \beta$  得到  $\gamma\alpha \rightarrow \gamma\beta$ , 再由传递律得到结论)。

• 属性集闭包:

对  $R(U, F), X \subseteq U, U = \{A_1, A_2, \dots, A_n\}$ , 令  $X_F^+ = \{A_i \mid \text{用 Armstrong 公理可从 } F \text{ 导出 } X \rightarrow A_i\}$  称  $X_F^+$  为  $X$  于  $F$  的属性集闭包。

显然  $X \subseteq X_F^+$ 。

引理:

$X \rightarrow Y$ , 当且仅当  $Y \subseteq X_F^+$ 。

• 求  $X$  的属性集闭包的算法:

1. 令  $X_0 = X$

2.  $B = \{A \mid (\exists V)(\exists W)(V \rightarrow W) \in F \wedge V \subseteq X_i \wedge A \subseteq W\}$  (对  $F$  中的任意元素  $V \rightarrow W$  检查, 如果  $V$  已在  $X_i$  中, 则把  $W$  的属性加入到  $X_{i+1}$  中)

3.  $X_{i+1} = B \cup X_i$

4. if  $X_{i+1} \neq X_i$  then:

$i = i + 1$

goto 2 (如果这轮检查仍有新的元素加入, 则下一次迭代也可能有未被检查过的元素加入, 否则算法结束)

5.  $X_F^+ = X_i$

• 覆盖:

对  $R(U)$  上的两个函数依赖集合  $F, G$ , 如果  $F^+ = G^+$ , 则称  $F$  和  $G$  是等价的, 也称  $F$  覆盖  $G$  或者  $G$  覆盖  $F$ 。

• 最小覆盖:

若  $F$  满足以下条件, 则称  $F$  为最小覆盖 (最小依赖集, 最小等价依赖集) :

- $F$  中每个函数依赖的右部是单个属性。
- 对任何  $\{X \rightarrow A\} \in F$ , 有  $F - \{X \rightarrow A\}$  不等价于  $F$ 。
- 对任何  $\{X \rightarrow A\} \in F$ ,  $Z \subset X$ ,  $(F - \{X \rightarrow A\}) \cup \{Z \rightarrow A\}$  不等价于  $F$ 。

• 求最小覆盖的算法:

1. 通过分解律「若  $\alpha \rightarrow \beta\gamma$  成立, 则  $\alpha \rightarrow \beta$  和  $\alpha \rightarrow \gamma$  成立」, 将原函数依赖集抄一遍, 右部分解为单属性。
2. 对任意  $X \rightarrow Y$ , 求  $X$  对于  $F$  剩余部分的属性闭包, 若不包含  $Y$ , 则  $X \rightarrow Y$  是必需的, 否则删除。

**Tips:**

对于仅在右部中出现一次的属性的那个依赖一定不会在这一步被删除, 因为不可能有其他依赖能推出它了。

3. 对于任意  $XY \rightarrow Z$ , 考虑能否替换为  $X \rightarrow Z$ 。求  $X$  在替换前依赖集的属性闭包, 如果不包含  $Z$ , 则替换是不可行的。如果也不能替换成  $Y \rightarrow Z$ , 则  $XY \rightarrow Z$  是必需的。

若  $X$  在替换前依赖集的属性闭包包含  $Z$ , 说明本来就有依赖  $X \rightarrow Z$ , 又因为  $XY \rightarrow X$ , 所以又可以推出  $XY \rightarrow Z$ , 即  $XY \rightarrow Z$  是多余的, 可以用  $X \rightarrow Z$  替换。

## 3.4. 关系模式范式

• 第一范式:

如果关系模式  $R$  所有属性的域都是原子, 则称  $R$  属于第一范式 ( $1NF$ ), 记作  $R \in 1NF$ 。

$1NF$  要求关系中不能有复合属性、多值属性及其组合。

多值属性解决方案:

1	Jones	Allan	2	555-1234	101	No
					108	Yes

- 拆成两个表, 原表去掉多值列, 然后把多值属性和主码作为一个新的表。
- 拆成多行
- 拆成多列

• 第二范式:

若  $R(U) \in 1NF$  且  $U$  中的每一非主属性完全函数依赖于候选键, 则称  $R(U)$  属于第二范式, 记为:  $R(U) \in 2NF$ 。

第二范式消除非主属性对候选键的部分依赖。

将候选键拆成多个表, 每个表的非主属性完全函数依赖于候选键。

第二范式只有历史意义, 已经不在实际中使用了。

• 第三范式:

若  $R(U, F) \in 2NF$ , 在  $R$  中若不存在候选键  $X$ , 属性集  $Y$  (不可以是候选键), 和非主属性  $Z$ , 使得  $X \rightarrow Y, Y \rightarrow Z$  成立, 其中  $Y \not\subseteq X, Z \not\subseteq Y, Z \not\subseteq X, Y \nrightarrow X$ , 则称  $R$  属于第三范式, 记为:  $R \in 3NF$ 。

第 3 范式消除了非主属性对候选键的传递依赖。

如满足第 3 范式，则一定能满足第 2 范式。

- Boyce-Codd 范式：

若  $R(U, F) \in 1NF$ , 若对于任何  $X \rightarrow Y \in F$  (或  $X \rightarrow A \in F$ ) , 当  $Y \not\subset X$  (或  $A \in X$ ) 时,  $X$  必含有候选键 (即超码) , 则称  $R(U)$  属于 Boyce-Codd 范式, 记为:  $R(U) \in BCNF$ 。

如满足 Boyce-Codd 范式，则一定能满足第 3 范式。

- 多值依赖:

对  $R(U)$ , 设  $X, Y \subset U$ , 若对于  $R(U)$  的任一关系  $r$ , 若元组  $t \in r, s \in r, t[X] = s[X]$ , 则必有  $u \in r, v \in r$  使得:

- $u[X] = v[X] = t[X] = s[X]$
- $u[Y] = t[Y]$  且  $u[U - X - Y] = s[U - X - Y]$
- $v[Y] = s[Y]$  且  $v[U - X - Y] = t[U - X - Y]$

$X$	$Y$	$U - X - Y$
$x$	$y_1$	$z_1$
$x$	$y_2$	$z_2$
$x$	$y_2$	$z_1$
$x$	$y_1$	$z_2$

均成立, 则称  $Y$  多值依赖于  $X$ , 或说  $X$  多值决定  $Y$ , 记作  $X \rightarrow\rightarrow Y$ , 同时  $X \rightarrow\rightarrow U - X - Y$ 。

若  $U = X \cup Y$ , 则一定有  $X \rightarrow\rightarrow Y$ 。

直观地, 若  $X \rightarrow\rightarrow Y$ , 对于  $X$  给定值,  $Y$  有一组值与之对应 (0 或  $n$  个) 且这组  $Y$  值不以任何方式与  $U - X - Y$  中属性值相联系。

- 第四范式 (基于多值依赖的 BCNF 范式, 不考) :

函数依赖和多值依赖集为  $F$  的关系模式  $R(U)$  属于第四范式 ( $4NF$ ) 的条件是, 对  $F^+$  中所有形如  $\alpha \rightarrow \beta$  的多值依赖 (其中  $\alpha \subseteq U$  且  $\beta \subseteq U$ ) , 至少有以下之一成立:

- $\alpha \rightarrow \beta$  是一个平凡的多值依赖。
- $\alpha$  是  $R$  的一个超码。

### 3.5. 模式分解

- 模式分解定义:

关系模式  $R(U)$  的分解是指用  $R$  的一组子集  $\rho = \{R_1, \dots, R_k\}$  来代替它。其中  $U = U_1 \cup U_2 \cup \dots \cup U_k, U_i \notin U_j (i \neq j)$ 。对于关系模式  $R$  的任一关系  $r$ , 它向  $\rho$  的投影连接记为  $m_\rho(r)$  :

$$m_\rho(r) = \pi_{R_1}(r) \bowtie \dots \bowtie \pi_{R_k}(r) = \bowtie_{(i=1, \dots, k)} \pi_{R_i}(r)$$

这里:  $\pi_{R_i}(r) = \{t[R_i] \mid t \in r, i = 1, \dots, k\}$ 。

若对关系模式  $R$  的任一关系  $r$  都有  $r = m_\rho(r)$  成立, 则称  $\rho$  是  $R$  相对于  $F$  的一个无损连接分解。

- 无损连接分解检验算法:

- 构造一  $k$  行  $n$  列的表 ( $k$  是分解的集合数,  $n$  是  $R(U)$  属性数), 可称为  $R_\rho$  表。
- 其中第  $j$  列对应于  $A_j$ , 第  $i$  行对应于  $R_i$ 。
- 若  $A_j \in R_i$ , 则  $R_\rho$  表中第  $i$  行第  $j$  列位置填写符号  $a_j$ , 否则填写  $b_{ij}$ 。
- 根据  $\forall(X \rightarrow Y) \in F$ , 对  $R_\rho$  表进行修改:

给定  $X \rightarrow Y$ , 寻找  $X$  属性取值相同的行, 用其值重置  $Y$  属性值。

- 修改后，如果有一行变成  $a_1, a_2, \dots, a_n$ ，则  $\rho$  是无损连接分解，否则为有损连接分解。

笔者认为这个算法应该循环，直到不能再修改为止，但是老师说原论文也没有提到做多几次的事情，只需要做一遍就行，遂躺平做一遍就好。

对于分解成两个子集的情况，可以快速判断的方法：

- 设  $F$  是关系模式  $R$  上的一个函数依赖集合， $\rho = \{R_1, R_2\}$  是  $R$  的一个分解。
- 则当且仅当  $R_1 \cap R_2 \rightarrow R_1 - R_2$  或者  $R_1 \cap R_2 \rightarrow R_2 - R_1$  属于  $F^+$  时， $\rho$  是关于  $F$  无损连接的。
- 因为  $R_1 = (R_1 \cap R_2) \cup (R_1 - R_2)$ ,  $R_2 = (R_1 \cap R_2) \cup (R_2 - R_1)$ ，所以有

	$R_1 \cap R_2$	$R_1 - R_2$	$R_2 - R_1$
$R_1$	$a_1$	$a_2$	$b_{13}$
$R_2$	$a_1$	$b_{22}$	$a_3$

- 此时只要  $R_1 \cap R_2 \rightarrow R_1 - R_2$  或者  $R_1 \cap R_2 \rightarrow R_2 - R_1$  属于  $F^+$  的一个成立，就能实现有一行变成  $a_1, a_2, a_3$ 。

- 保持依赖分解的分解  $\chi$ ：

- 对于关系模式  $R(U, F)$ ,  $U$  是属性全集,  $F$  是函数依赖集合,  $\rho = \{R_1, \dots, R_k\}$  是  $R$  的一个分解。
- 如在  $\pi_{R_i}(F)$  中的所有依赖之并集 ( $i = 1, \dots, k$ )，逻辑蕴涵  $F$  的每个依赖，则称分解  $\rho$  保持依赖集  $F$ 。
- 其中  $\pi_{R_i}(F)$  是  $F$  在  $R_i$  上的投影，即  $F$  中的任一投影  $X \rightarrow Y$ ，如果  $X, Y \in R_i$ ，则  $X \rightarrow Y \in \pi_{R_i}(F)$ 。

- 保持依赖分解检测算法：

```
对  $F$  中的每一个  $\alpha \rightarrow \beta$ ,
result  $\leftarrow \alpha$ 
while
  foreach 分解后包含有  $\alpha$  的  $R_i$  :
     $t \leftarrow (\text{result} \cap R_i)^+ \cap R_i$ 
    result  $\leftarrow \text{result} \cup t$ 
until ( result 没有变化 )
如果 result 含  $\beta$  中的所有属性，则函数依赖  $\alpha \rightarrow \beta$  保持。
```

- BCNF 分解 + 无损连接分解：

BCNF 范式要求每个函数依赖的左侧都要是关系模式的超码（含有候选键）。

- 仅 BCNF 分解：

将左侧不含候选键的函数依赖单独组成一个关系，将包含候选键的组成一个关系。

- 无损连接分解成 BCNF：

```
计算  $F^+$ 
while result  $\notin \text{BCNF}$  do :
  if result 中存在模式  $R_i$  不属于 BCNF then:
    对于  $R_i$  上所有非平凡函数依赖  $\alpha \rightarrow \beta$  (满足  $\alpha \rightarrow R_i$  不属于  $F^+$ ，即  $\alpha$  不是  $R_i$  的超码)
      result = (result -  $R_i$ )  $\cup$  ( $R_i - \beta$ )  $\cup$  ( $\alpha, \beta$ )
    // 即用两个模式  $(R_i - \beta) \cup (\alpha, \beta)$  取代原来的  $R_i$ ，由之前的快速判断的方法可知分解无损
```

- 3NF 分解 + 保持依赖分解 (+ 无损连接分解)：

- 仅 3NF 分解：

将每一个函数依赖单独组成一个关系。

- 保持依赖分解成 3NF:

▪ 关系模式  $R(U, F)$ ,  $F$  是函数依赖集最小覆盖，求保持依赖的 3NF 分解  $\rho$ 。

- 若有  $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_m \in F$ , 则以  $XA_1 A_2 \dots A_m$  组成一模式  $R_i$ ,  $\rho = \rho \cup R_i$

这样保证  $R_i$  是  $3NF$  分解:

因为  $F$  是最小覆盖, 要求对任何  $\{X \rightarrow A\} \in F$ , 有  $F - \{X \rightarrow A\}$  不等价于  $F$ 。

所以一定不会存在  $A_i \rightarrow A_j \in F (i \neq j)$  的情况, 否则  $F - \{X \rightarrow A_j\}$  仍等价于  $F$ 。

- 上一步处理完后, 把  $R$  中不出现在  $F$  中的属性去掉并单独组成一模式  $R_{no}$ , 令  $\rho = \rho \cup R_{no}$ 。

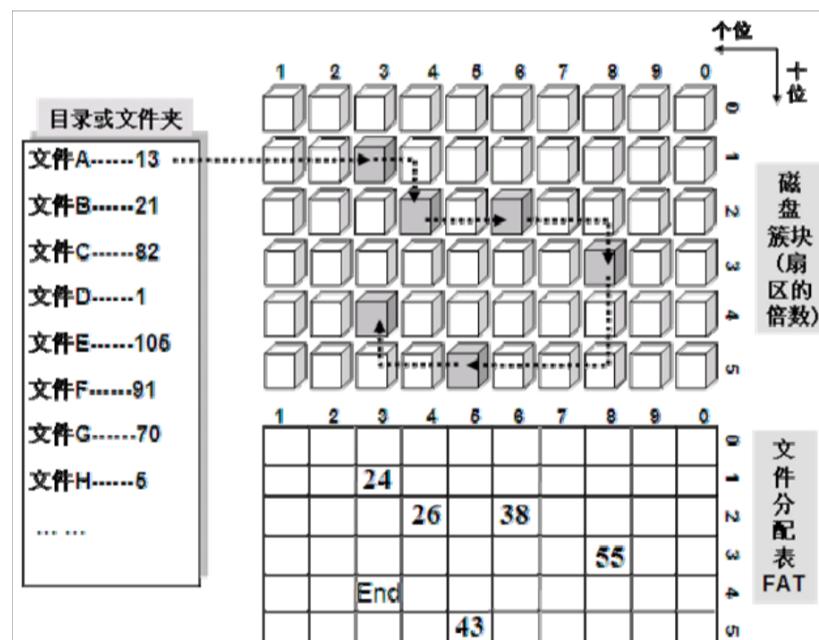
若要达到无损连接分解:

- 如果  $R_{no}$  存在, 将在  $R_{no}$  加入候选键属性, 则能使该分解达到无损。
- 如果不存在, 则判断所得  $\rho$  集合中的是否存在一个  $R_i$  包含了候选键属性集:
  - 存在, 算法结束。
  - 不存在,  $\rho = \rho \cup$  候选键属性集。

## 4. 数据存储和查询

### 4.1. 存储和文件结构

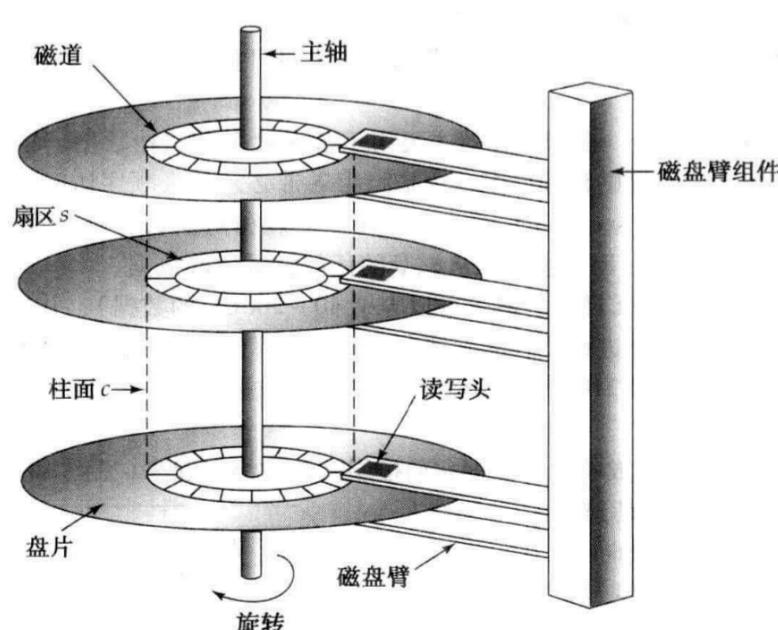
- 存储体系:
  - 将不同性价比的存储器组织在一起, 满足高速度、大容量、低价格需求。
  - CPU 与内存直接交换信息, 按存储单元 (存储字) 进行访问。
  - 外存按存储块进行访问, 其信息需先装入内存, 才能被 CPU 处理。
- 操作系统对数据的组织:
  - FAT (文件分配表 - File Allocation Table) - 目录 (文件夹) - 磁盘块/簇



- 对于一个文件, 用 FAT 找到它在磁盘中的位置。

- 内存管理:
 

一条记录的地址 = 存储单元的地址 = 内存地址 = 页面 : 页内偏移量
- 磁盘结构:



- 盘片的表面从逻辑上划分为磁道（同心圆），磁道又划分为扇区。
- 扇区是从磁盘读出和写入信息的最小单位，系统数据传输的基本单位是磁盘块（几个连续的扇区）。
- 一个磁盘的基本信息（假设）：
  - 8个圆盘，16个盘面
  - 每个盘面有 $2^{16}$ 或65536个磁道
  - 每个磁道（平均）有 $2^8 = 256$ 个扇区
  - 每个扇区有 $2^{12} = 4096$ 个字节
  - 磁盘的容量 =  $2^4 \times 2^{16} \times 2^8 \times 2^{12} = 2^{40}$ 字节
- 磁盘数据读写时间：
  - 包括寻道时间（约在 $1 - 20ms$ ），旋转时间（约 $0 - 10ms$ ）和传输时间。
  - 磁盘以7200转/min旋转，即 $8.33ms$ 内旋转一周
  - 柱面之间移动磁头组合从启动到停止花费 $1ms$ 。
  - 每移动4000个柱面另加 $1ms$ ，即磁头在 $0.00025ms$ 内移动一个磁道，从最内圈移动到最外圈，移动65536个磁道大约用 $0.00025 \times 65536 + 1 = 17.38ms$ 。
  - 一个磁道中扇区间的空隙大约占10%的空间
  - 一个磁盘块 = 4个扇区 = 16384个字节
  - 最短时间 = 传输时间大约是 $0.13ms$
  - 最长时间 = 寻道时间+旋转时间+传输时间 =  $17.38 + 8.33 + 0.13 = 25.84ms$
  - 平均时间 =  $6.46(16.38/3 + 1) + 4.17(8.33/2) + 0.13 = 10.76ms$ （括号表示这个数是怎么得到的）

概率论忘掉了，这里可能有算错的地方（这是不会考的，不重要）

平均寻道时间要除以3是求了一个盘片内径向任意两点之间移动时间的期望（前提假定为平均分布）。

设盘片半径为 $r$ 。令 $f(y) = |x - y|$ ，即表示径向从 $y$ 点到 $x$ 点之间距离。

当 $x$ 为 $[0, r]$ 上的固定值时，求出 $[0, r]$ 上所有点到 $x$ 的期望距离：

$$\mathbb{E}f(y) = \int_0^r |x - y| \frac{1}{r} dy = \frac{x^2}{r} - x + \frac{r}{2}$$

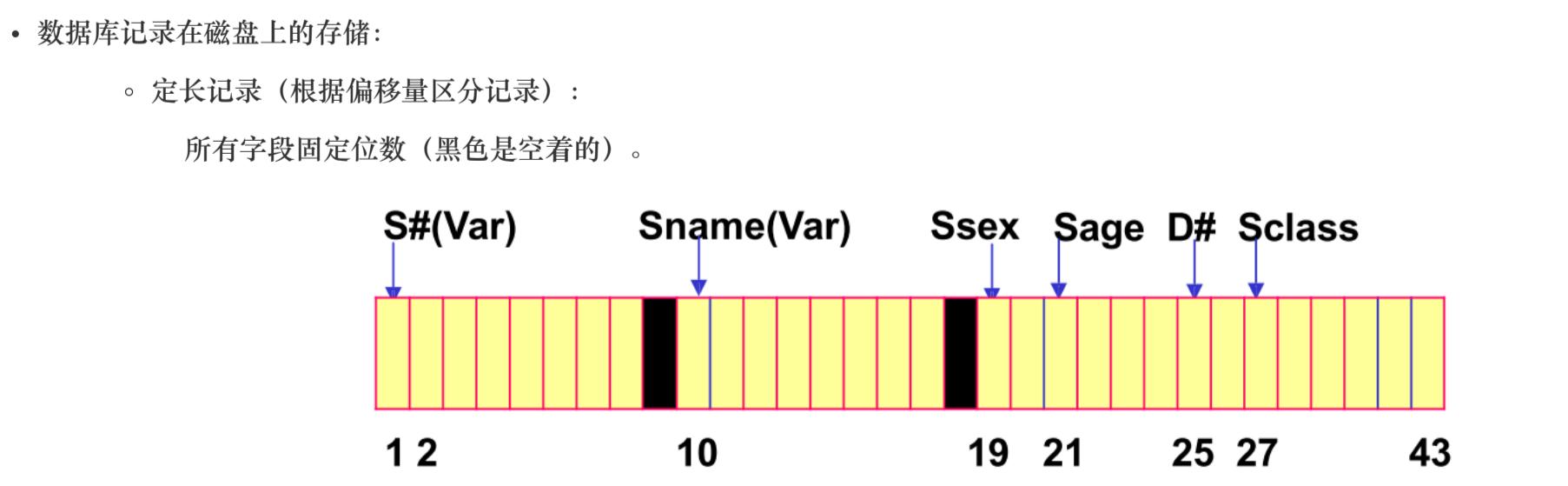
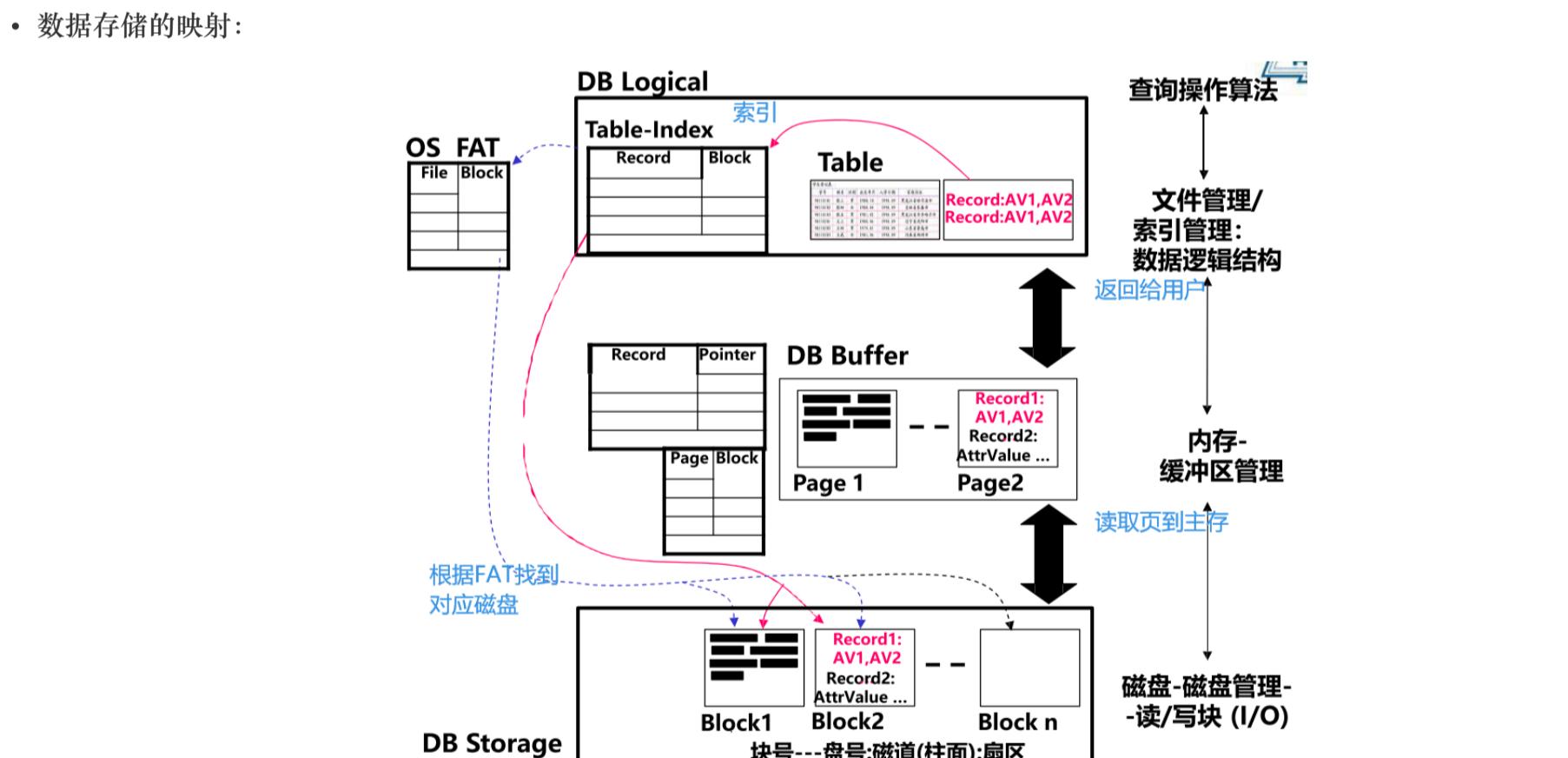
对于所有 $x \in [0, r]$ ，计算期望距离的期望：

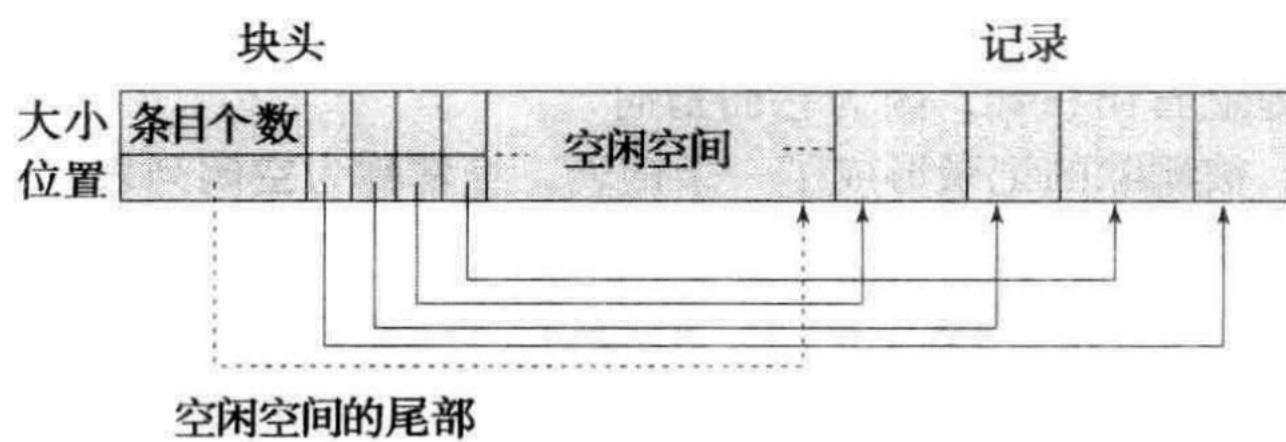
$$\mathbb{E}\mathbb{E}f(y) = \int_0^r \frac{1}{r} \left( \frac{x^2}{r} - x + \frac{r}{2} \right) dx = \frac{r}{3}$$

即磁头平均会移动 $\frac{r}{3}$ 。

- 物理存取算法考虑的关键：
  - 降低I/O次数。
  - 降低排队等待时间。
  - 降低寻道/旋转延迟时间：
    - 同一磁道连续块存储
    - 同一柱面不同磁道并行块存储
    - 多个磁盘并行块存储
- 独立磁盘冗余阵列（Redundant Array of Independent Disk, RAID技术）（理解）：
  - 块级拆分：
    - 一个文件由多个块组成，不同块存储于不同磁盘。
  - 比特级拆分：
    - 一个字节被拆分成8个比特位，不同比特位存储于不同磁盘。
  - RAID 0级：
    - 块级拆分但没有任何冗余（例如镜像或奇偶校验位）的磁盘阵列。
  - RAID 1级：
    - 使用块级拆分的磁盘镜像，每一个磁盘有一个镜像磁盘。

- RAID 2 级：  
位交叉纠错处理，4个磁盘存储4位 + 3个校验盘存储3校验位（汉明码）。
- RAID 3 级：  
RAID 2 级的改进，只用一个校验盘，比 RAID 2 级常用。
- RAID 4 级：  
块交叉的奇偶校验组织结构。它像 RAID 0 级一样使用块级拆分，此外在一张独立的磁盘上为其他  $N$  张磁盘上对应的块保留一个奇偶校验块。
- RAID 5 级：  
块交叉的分布奇偶校验的组织结构，是 RAID 4 级的改进，将数据和奇偶校验位都分布到所有的  $N + 1$  张磁盘中。





- 块头包括：
    - 块头中记录条目的个数。
    - 块中空闲空间的末尾处。
    - 一个由包含记录位置和大小的记录条目组成的数组。
  - 实际记录从块的尾部开始连续排列。
  - 块中空闲空间是连续的，无论是插入操作还是删除操作都不能改变这一点。
  - 如果插入一条记录，在空闲空间的尾部给这条记录分配空间，并且将包含这条记录大小和位置的条目添加到块头中。
  - 如果一条记录被删除：
    - 它所占用的空间被释放，并且它的条目被设置成被删除状态。
    - 块中在被删除记录之后的记录将被移动，使得由删除而产生的空闲空间被重用，并且所有空闲空间仍然保持连续。
    - 块头中的空闲空间末尾指针要做适当修改。

移动记录的代价并不高，因为块的大小是有限制的，典型的值为 4 – 8KB。

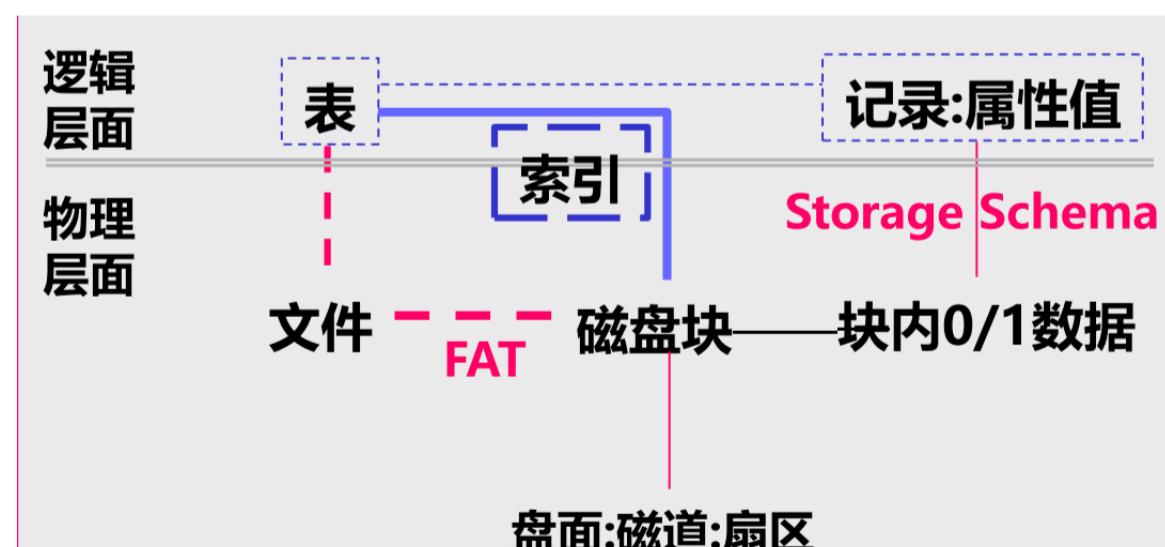
- 数据库 - 表所占磁盘块的分配方法:
    - 连续分配:

数据块被分配到连续的磁盘块上（会存在扩展困难问题）。
    - 链接分配:

数据块中包含指向下一数据块的指针（访问速度问题），不连续，有空位就放。
    - 按簇分配:

按簇分配，簇内连续分配，簇之间靠指针连接，簇有时也称片段，结合前面两者优点。
    - 索引分配:

索引块中存放指向实际数据块的指针（可以不用 FAT）。



- ## • 文件组织方法

考试重点，注意区分 ☺，回答合理即可。

- 无序文件组织：
    - 无序记录文件（堆文件 heap 或 pile file）
      - 记录可存储于任意有空间的位置，磁盘上存储的记录是无序的。更新效率高，但检索效率可能低。

- 一开始新记录总插入到文件尾部。删除记录时，可以直接删除该记录所在位置的内容，也可以在该记录前标记「[删除标记](#)」，新增记录可以利用那些标记为「[删除标记](#)」的记录空间
- 频繁删增记录时会造成空间浪费，所以需要周期性[重新组织数据库](#)。

数据库重组是通过移走被删除的记录使[有效记录连续存放](#)，从而回收那些由删除记录而产生的未利用空间（外部碎片）。

#### ◦ 有序文件组织

- 有序记录文件（排序文件 **Sequential**）
- 记录按某属性或属性组值的顺序插入，磁盘上存储的记录是有序的。[检索效率可能高](#)，但更新效率低。

当按排序字段进行检索时，速度得到很大提高。但当按非排序字段检索时，速度可能不会提高很多。

- 用于存储排序的属性通常称为[排序字段](#)，可以是关系中的[主码](#)，所以又称[排序码](#)。
- 改进办法（使用溢出）：
  - 为将来可能插入元组预留空间（这可能造成空间浪费），或使用一个[临时的无序文件](#)（被称为[溢出文件](#)）保留新增的记录。
  - 当采取溢出文件措施时，检索操作既要操作主文件，又要[操作溢出文件](#)。
  - 需要周期性[重新组织数据库](#)，将溢出文件[合并](#)到主文件，并恢复主文件中的记录顺序。

#### ◦ 散列文件组织：

- 散列文件（**Hash file**）
- 可以把记录按某属性或属性组的值，依据一个[散列函数](#)来计算其应存放的[位置（桶号）](#)，检索效率和更新效率都有一定程度的提高。
- 用于进行散列函数计算的属性通常称为[散列字段](#)，散列字段通常也采用关系中的[主码](#)，所以又称[散列码](#)。
- 不同记录可能被 **hash** 成同一桶号，此时需在桶内顺序检索出某一记录。
- 答题万金油，读写性能都不错。

#### ◦ 聚簇文件组织：

- 聚簇文件（**Clustering file**）
- 聚簇：
  - 将具有[相同或相似属性值](#)的记录存放于[连续](#)的磁盘簇块中，优化连接代价，在[不用索引](#)的时候使用。
  - 多表聚簇：
    - 将若干个[相互关联](#)的表存储于一个文件中，可提高多表情况下的查询速度。
    - 何时使用多表聚簇依赖于数据库设计者所认为的[最频繁的查询类型](#)。

## 4.2. 索引概念和分类

### ◦ 索引定义：

- 定义在存储表基础上，无需检查所有记录，快速定位所需记录的一种辅助存储结构，由[一系列](#)存储在[磁盘](#)上的[索引项](#)组成，每一索引项又由两部分构成：
  - 索引字段**：  
由表中[某些列](#)中的值串接而成，类似于词典中的词条。索引中[通常](#)存储了索引字段的每一个值。
  - 行指针**：  
指向表中包含索引字段值的记录在磁盘上的[存储位置](#)，类似于词条在书籍、词典中出现的页码。
- 有索引时，更新操作必须[同步更新](#)索引文件和主文件。
- 对经常出现在[检索条件](#)、[连接条件](#)和[分组计算条件](#)中的属性可建立索引。

存储索引项的文件为[索引文件](#)，存储表称为[主文件](#)。

### ◦ 索引文件组织方式有两种：

- 排序索引文件**：  
按索引字段值的某一种[顺序](#)组织存储。

- 散列索引文件:

依据索引字段值使用散列函数分配散列桶的方式存储。

**主文件组织**有堆文件、排序文件、散列文件、聚簇文件等多种方式，和**索引文件组织方式**区分。

- 索引应用的评价:

- 访问时间:

在查询中使用该技术找到一个特定数据项或数据项集所需的时间

- 插入时间:

插入一个新数据项所需的时间。该值包括**找到**插入这个新数据项的正确位置所需的时间，以及**更新索引结构**所需的时间。

- 删除时间:

删除一个数据项所需的时间。该值包括找到待删除项所需的时间，以及更新索引结构所需的时间。

- 空间开销:

索引结构所占用的额外存储空间。倘若存储索引结构的额外空间大小适度，通常牺牲一定的空间代价来换取性能的提高是值得的。

- 码的区分 (⌚不太能区分) :

- 排序码:

对主文件进行排序存储的那些属性或属性组。

- 索引码:

即索引字段，**不一定具有唯一性**。

- 搜索码:

在文件中查找记录的属性或属性集称为**搜索码**。

- 稠密索引:

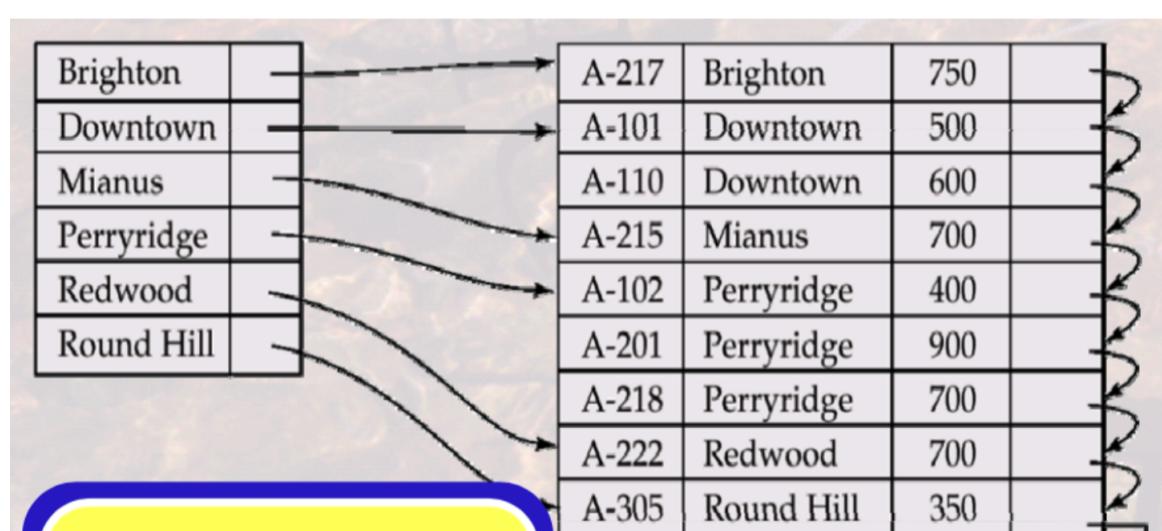
- 主文件中每一个**搜索码值**（索引字段值）都有一个索引项和它对应。

- 对**候选键**建稠密索引:

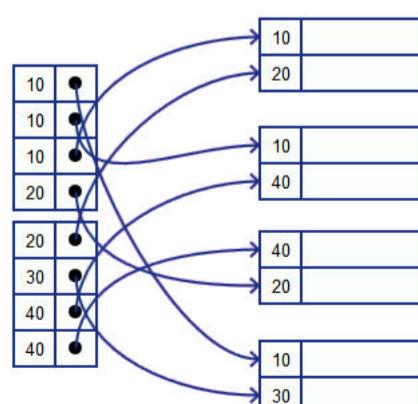
主文件不用排序，直接可以定位。

- 非候选键建稠密索引:

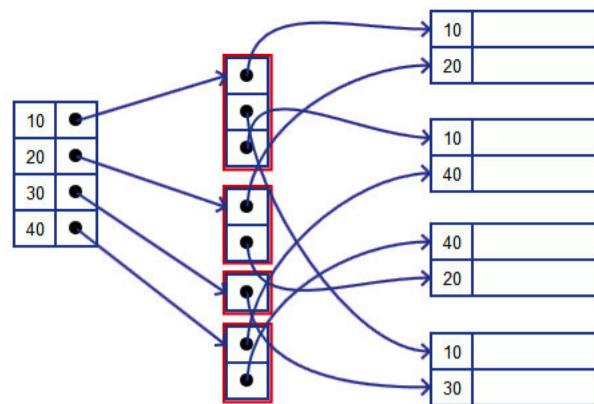
- **索引文件**中索引字段值是**不重复的**，**主文件**按索引字段**排序**。



- **索引文件**中索引字段值是有**重复的**，**主文件****不排序**。



- 引入**指针桶**处理非候选键索引的多记录情况，**索引文件**中索引字段值是**不重复的**，**主文件****不排序**。



索引文件通常要排序。

- 稀疏索引：

稀疏索引只为主文件部分搜索码值（索引字段值）建立索引记录，主文件按索引字段排序。

- 主索引：

- 如果包含记录的文件按照某个搜索码指定的顺序排序，那么该搜索码对应的索引称为聚集索引，也称为主索引。
- 主索引对每一存储块有一个索引项，索引项的总数和存储表所占的存储块数目相同。
- 存储表的每一存储块的第一条记录，又称为锚记录，或简称为块锚。
- 主索引的索引字段值为块锚的索引字段值（通常为主码值或排序码值），而指针指向其所在的存储块。
- 主索引（可以）是稀疏索引。

- 辅助索引：

- 辅助索引定义在主文件的任一或多个非排序字段上的辅助存储结构。
- 辅助索引一定是稠密索引。

主索引和辅助索引的区别：

- 一个主文件仅可以有一个主索引（只按主索引的搜索码排序），但可以有多个辅助索引。
- 主索引通常建立于主码/排序码上面，辅助索引建立于其他属性上面。
- 可以利用主索引重新组织主文件数据，辅助索引不能改变主文件数据。
- 主索引是稀疏索引，辅助索引是稠密索引

- 聚簇索引：

- 聚簇索引定义：

聚簇索引是指索引中邻近的记录在主文件中也是临近存储的。

- 聚簇字段定义：

如果主文件的某一排序字段不是主码，则该字段上每个记录取值便不唯一，此时该字段被称为聚簇字段。聚簇索引通常是在聚簇字段上定义的。

T#是“非主码字段”，  
有重复值，排序存储

聚簇索引文件

T#
001
003
004

聚簇字段值 块指针

主文件

C#	Cname	Chours	Credit	T#
001	数据库	40	6	001
004	编译原理	40	6	
009	数理逻辑	40	6	
008	组合数学	30	4.5	003
003	数据结构	40	6	
005	C语言	30	4.5	
006	计算机原理	40	6	004
002	高等数学	80	12	
007	外语	80	12	

主文件基于  
“非主码的  
字段”T#排  
序存储

- 非聚簇索引：

指索引中邻近的记录在主文件中不一定是邻近存储的。

聚簇索引和非聚簇索引的区别：

- 一个主文件只能有一个聚簇索引文件，但可以有多个非聚簇索引文件。

- 主索引通常是聚簇索引（但其索引项总数不一定和主文件中聚簇字段上不同值的数目相同，其和主文件存储块数目相同）。
- 辅助索引通常是非聚簇索引。
- 主索引/聚簇索引是能够决定记录存储位置的索引，而辅助索引/非聚簇索引则只能用于查询，指出已存储记录的位置。

## 4.3. 多级索引 (B+ 树)

必考内容

当索引项比较多时，可以对索引再建立索引，依此类推，形成多级索引。

- B+ 树节点：

$P_1$	$K_1$	$P_2$	...	$P_{n-1}$	$K_{n-1}$	$P_n$
-------	-------	-------	-----	-----------	-----------	-------

- $K_i$ ：索引字段值
- $P_i$ ：
  - 是指向 索引块或数据块 或数据块中记录 的指针：
    - 非叶结点指针指向索引块。
    - 叶结点指针指向主文件的数据块（稀疏索引）或数据记录（稠密索引）。
    - 每个索引块的指针利用率（书上是说索引字段值的利用率）都在 50% – 100% 之间（根节点可以不满足）。
    - 索引字段值  $x$  在  $K_{i-1} \leq x < K_i$  的由  $P_i$  指向（设  $K_0 = 0$ ），而  $K_i \leq x < K_{i+1}$  的由  $P_{i+1}$  指向。
    - 叶节点的  $P_n$  指向下一个叶节点。
  - 共有  $n - 1$  个索引字段值和  $n$  个指针。

- B 树和 B+ 树比较：

- B 树索引字段值仅出现一次，可以在叶结点或者在非叶结点；B+ 树有重复索引字段值。
- B 树指向主文件的指针出现于叶结点或非叶结点，另一个指针指向结点；B+ 树只有叶子结点有指向主文件的指针。
- B 树所有结点才能覆盖所有键值的索引；B+ 树叶子结点就能覆盖。
- B 书叶节点和 B+ 数叶节点结构相同。

- B+ 树插入：

- 在原树的叶子节点中找到应该插入的位置：
  - 当索引块满时，需要分裂，分裂后也要保证指针利用率不低于一半（平均分）。

将这  $n$  个搜索码值（叶结点中原有的  $n - 1$  个值再加上新插入的值）分为两组，前  $\lceil n/2 \rceil$  个放在原来的结点中，剩下的放在一个新结点中。

- 否则直接插入，如果插入位置在叶子索引块最左边（即小于该块的其他索引块，要修改父节点的值）。
- 分裂可能引发连锁反应，由叶结点直至根结点判断。
- 分裂后需要仔细调整索引键值及指针。
- 注意叶子结点之间链接关系的调整。

- B+ 树删除：

- 当删除后指针利用率低于一半，先考虑从相邻叶子结点借索引项，如果不能借就合并。
- 合并可能引发连锁反应，由叶结点直至根结点（父节点索引值少了一个）。
- 合并后需要仔细调整索引键值及指针。
- 注意叶子结点之间链接关系的调整。

## 4.4. 散列索引

- 桶：

表示能存储一条或多条记录的一个存储单位，一个桶可以是一个或者若干个连续的存储块。

- 散列函数:

$K$  表示所有搜索码值的集合,  $B$  表示所有桶地址的集合, 散列函数  $h$  是一个从  $K \rightarrow B$  的函数。

散列函数要满足分布是均匀的和随机的。

- 溢出桶:

如果一条记录必须插入桶  $b$  中, 而桶  $b$  已满, 系统会为桶  $b$  提供一个溢出桶, 并将此记录插入到这个溢出桶中。

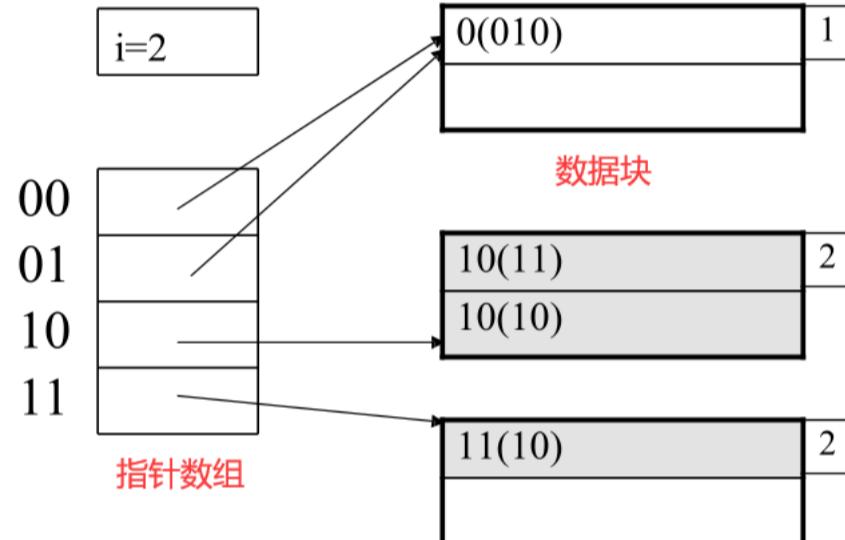
- 静态散列索引的缺点:

如果桶的数目  $M$  不变, 过大则浪费, 过小则将产生更多的溢出桶, 增加散列索引检索的时间。

以下为两种动态散列索引, 考区别。

- 可扩展散列索引:

- 有一个附加的间接层 (指针数组), 系统在访问桶本身之前必须先访问指针数组表。
- 指针数组能增长, 其长度总是 2 的幂。因而数组每增长一次, 桶的数目就翻倍。
- 多个桶可能共享一个数据块 (即多个指针数组指向同一个数据库)。



- 参数  $k$  表示散列函数所可能使用的最多位数。

- 散列函数:

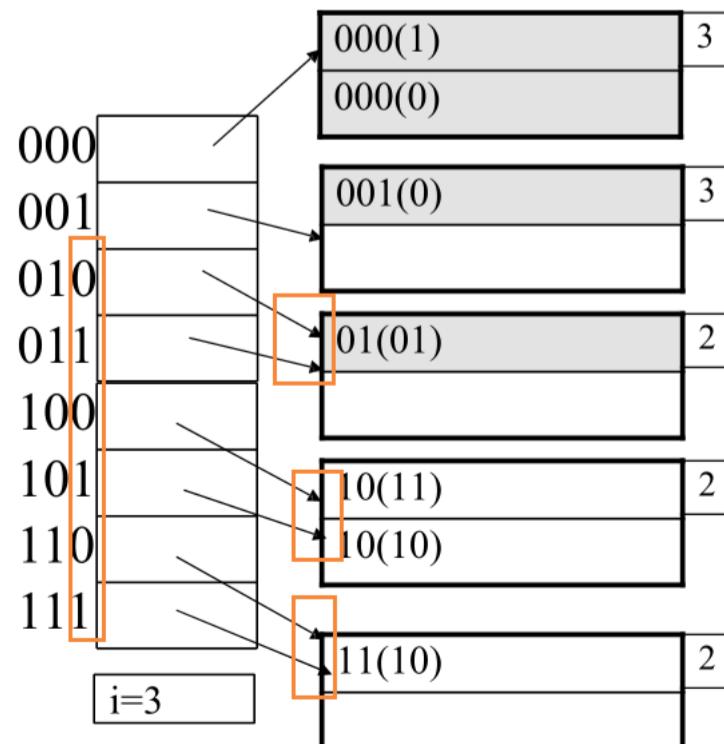
散列函数  $h$  为每个键计算出一个  $k$  位二进制序列 (散列函数值)。

- $i$  为散列函数当前已经使用到的最多位数。取散列函数值的前  $i$  位 (散列前缀) 为作为要插入桶的编号。

- 插入某个桶时发现已满, 则需要扩展散列桶, 进行分裂:

- $i$  增 1, 重新散列该块的数据到两个块中。

- 其他的桶按照散列前缀重新指向对应数据块, 会出现多个桶共享一个块的情况 (因为在后面加 0/1 指向不变)。



缺点:

- 翻倍要处理的工作量很大。
- 桶数翻倍后，主存可能装不下。
- 可能产生大量空间浪费，因为一个数据块满了得翻倍，其他块可能都没有存多少数据。

- 线性散列索引：

- 桶数的选择：

保持与存储块所能容纳的记录总数成一个**固定的比例**，例如 80%。超过此比例，则桶数**仅增长 1 块**。不超过这个比例时允许有溢出桶。

- 假定散列函数值的  $i$  位为桶数组项编号。现在要插入一个键值为  $K$  的记录：

- 通过散列函数得知要插入编号为  $a_1a_2\dots a_i$  的桶中，即  $a_1a_2\dots a_i$  是  $h(K)$  的后  $i$  位。设  $m = a_1a_2\dots a_i$ ,  $n$  为当前的桶数。

- 如果  $m < n$ , 则编号为  $m$  的桶存在，并把记录存入该桶中。

- 如果  $n \leq m < 2^i$ , 那么桶还不存在，因此我们把记录存入桶  $m - 2^{i-1}$ , 也就是当我们**把  $a_1$  (它现在是 1) 改为 0 时对应的桶**。

- 如果插入后不满足比例时，要分裂一个桶成两个，这个要分裂的桶的编号是这样确定的：

设当前  $n$  已增一，则分裂与  $n$  **低位相同而最高位不同的那一个桶**，即假设当前为  $n = 1a_2a_3\dots a_i$ , 则就从  $0a_2a_3\dots a_i$  对应的桶分裂而来。

因为当时有些记录是强行插入桶中的（上面  $n \leq m < 2^i$  的情况，那时还没有  $1a_2a_3\dots a_i$  这个桶，也得硬插），插入的桶号就是  $0a_2a_3\dots a_i$ , 现在分裂了，就可以插入对应正确的桶中了。

- 分裂只会影响分裂之后的两个桶的记录，其他保持不变。

- 当桶数超过  $2^i$  个桶时，则使  $i$  增 1。

## 5. 数据库查询实现算法

### 5.1. 概述

- 查询优化：

构造具有最小查询执行代价的查询执行计划应当是系统的责任，这项工作叫作查询优化：

- 利用模型的语义及完整性规则，优化查询。
- 去掉无关的表
- 去掉无关的属性
- 改写成等价的效果更好的语句

- 逻辑层优化：

优化关系代数操作执行次序，**后面会详细讲**。

- 物理层优化：

优化关系代数操作实现算法，存取路径与执行次序。

为每一个关系代数操作选择优化的执行例行程序，形成**物理查询计划**。

- 物理层优化的查询实现：

- DBA 获取数据库的相关信息（**定期统计**）。
- 选择相应操作的例行程序
- 依据相关信息进行**代价估算**，并选择代价最少的例行程序及确定相应的参数。
- 形成查询计划，以基本的例行程序为基本步骤，确定这些例行程序的执行顺序。

- 一次单一元组的一元操作（选择和投影）：

- 选择  $\sigma_F(R)$
- 投影  $\pi_\alpha(R)$

- 整个关系的一元操作：

- 去重  $\delta(R)$ , DISTINCT
- 聚集  $\gamma(R)$ , GROUPBY
- 排序  $\tau(R)$ , SORTING

- 整个关系的二元操作:

- 集合上的操作:

$\cup_S, \cap_S, -S$

- 包 (允许重复的集合) 上的操作:

$\cup_B, \cap_B, -B$

- 笛卡尔积, 连接:

PRODUCT, JOIN

## 5.2. 连接操作的实现算法

- 关系的物理存储相关的参数:

关系是存储在磁盘上的, 磁盘是以磁盘块为操作单位, 首先要被装载进内存 (I/O 操作), 然后再进行元组的处理。

- $T_R$ :

关系 R 的元组数目。

- $B_R$ :

关系 R 的磁盘块数目。

- $M$ :

主存缓冲区的页数 (主存每页容量等于一个磁盘块的容量)。

- $I_R$ :

关系 R 的每个元组的字节数。

- $b$ :

每个磁盘块的字节数。

$$B_{R \times S} = T_R T_S (I_R + I_S) / b$$

下面讨论各种连接操作的实现算法 :

- 因为 I/O 操作所需时间远大于内存操作, 下面仅考虑 I/O 用时 (次数)。
- 忽略写回结果所需时间 (也就是说只考虑 I(input) 没有 O(output)), 因为很难估计。

- 连接操作算法  $P_1$  (主存利用率低) :

```

1  /* I/O 操作 */
2  For i = 1 to BR
3      read i-th block of R      // 执行 BR 次
4      For j = 1 to BS
5          read j-th block of S  // 执行 BR * BS 次
6
7      /* 内存操作 */
8      For p = 1 to b/IR        // 取元组, b/IR 表示一个磁盘块有多少个关系 R 的元组
9          read p-th record of R
10         For q = 1 to b/IS
11             read q-th record of S
12             if R.A 关系 S.B then
13                 串接 p-th record of R 和 q-th record of S;
14                 存入结果关系;

```

I/O 次数估计为  $B_R + B_R \times B_S$

- 连接操作的全主存实现算法  $P_2$ :

```

1 /* 算法假定内存大小 M >= BR + BS, 即只需要读一遍关系 R 和 S 的磁盘块 */
2 For i = 1 to BR
3     read i-th block of R
4 For j = 1 to BS
5     read j-th block of S
6 /* 内存操作稍有不同, 不是重点 */

```

I/O 次数估计为  $B_R + B_S$

- 连接操作的半主存实现算法  $P_3$ :

```

1 /* 算法假定内存大小 min(BR,BS) < M < BR + BS, 这里假设较小的是 BR, 读一次 R 的磁盘块放内存, S 的每次读就处理, 不用全部放入内存再处理 */
2 For i = 1 to BR
3     read i-th block of R
4 For j = 1 to BS //一次读入一块关系 S 的磁盘块
5     read j-th block of S
6 /* 内存操作稍有不同, 不是重点 */

```

I/O 次数估计为  $B_R + B_S$

- 连接操作的大关系实现算法  $P_4$  (将主存充分利用) :

```

1 /* 把关系 R 划分为 BR/(M-2) 个子集合, 每个子集合具有 M-2 块。令 MR 为 M-2 块容量的主存缓冲区,
2 * MS 为 1 块容量的 S 的主存缓冲区, 还有 1 块作为输出缓冲区。
3 */
4 For i = 1 to BR/(M-2) //一次读入M-2块
5     read i-th Sub-set of R into MR      // 执行 BR/(M-2) 次, 总共 BR 次 I/O
6     For j = 1 to BS //一次读入一块
7         read j-th block of S into MS    // 执行 BR/(M-2) * BS 次
8         For p = 1 to (M-2)b/IR
9             /* 内存操作稍有不同, 不是重点 */

```

I/O 次数估计为  $B_S(B_R/(M-2)) + B_R$

迭代器不考。

### 5.3.一趟扫描算法

- 关系  $R$  数据读取:

$B(R)$  是  $R$  的存储块 (磁盘块) 数目

$T(R)$  是  $R$  的元组数目

忽略写回代价

◦ 聚簇关系:

指关系的元组集中存放 (一个块中仅是一个关系中的元组)。

- **TableScan( $R$ )** 为表空间扫描算法:

扫描结果未排序  $B(R)$

- **SortTableScan( $R$ )**:

扫描结果排序  $B(R)$  or  $3B(R)$

内存装得下  $R$  就只需要  $B(R)$ , 否则需要  $3B(R)$ :

- 先读一遍  $R$ , 做第一趟排序。
- 写回磁盘一次, 因为还需要第二趟排序。
- 再多路读  $R$ , 做归并排序。

- 不考虑写回，上面共  $3B(R)$ 。

- **IndexScan(R)** 为索引扫描算法：

扫描结果未排序  $B(R)$

- **SortIndexScan(R)**：

扫描结果排序  $B(R)$  or  $3B(R)$

- 非聚簇关系：

关系的元组不一定集中存放（一个块中不仅是一个关系中的元组）。最坏情况下就是相邻的元组都不在同一磁盘块，每次都要读一个新磁盘。

- 扫描结果未排序为  $T(R)$

- 扫描结果排序为  $T(R) + 2B(R)$

- 去重复  $\&(R)$ ：

- 在内存中保存已处理过的元组。
- 当新元组到达时，需与之前处理过的元组进行比较。
- 建立不同的内存数据结构（排序结构/散列结构/B+ 树），来保存之前处理过的数据，以便快速判断是否重复。
- 算法复杂性为  $B(R)$
- 应用条件为  $B(\&(R)) \leq M$

- 分组聚集  $\gamma_L(R)$

- 需要在内存中保存所有的分组，保存每个分组上的聚集信息。
- 建立不同的内存数据结构（排序结构/散列结构/B+ 树），来保存之前处理过的数据，以便快速处理整个关系上的操作。
- 算法复杂性为  $B(R)$
- 应用条件为所有分组的数量应能在内存中完整保存。

- 集合或者包上的二元运算：

- 扫描一个关系  $R$ ，然后再扫描另一个关系  $S$ 。
- 集合的操作需要去重复。包的操作需要做计数每个元组的出现次数（具体操作还需具体分析）。
- 算法复杂性为  $B(R) + B(S)$
- 应用条件为  $\min(B(R), B(S)) \leq M$ （只要用一个能放入内存就行）。

- 连接操作实现算法  $P_4$  的改进：

- 主要思想就是把内存操作的两重循环通过散列的方式，压缩到一重循环就行。
- 散列函数可取连接条件中的相应属性，使得散列结果相同说明这两个属性满足连接条件。

## 5.4. 基于索引的选择算法

- 选择条件中有涉及到索引属性时，可以使用索引，辅助快速检索。
- 聚簇和非聚簇索引，使用时其效率是不一样的。
- 案例分析：

  - 假设  $B(R) = 1000, T(R) = 20000$ ，即  $R$  有 20000 个元组存放到 1000 个块中。
  - $a$  是  $R$  的一个属性，在  $a$  上有一个索引，并且考虑  $\sigma_{a=0}(R)$  操作：
    - 如果  $R$  是聚簇的，且不使用索引，查询代价 = 1000 个 I/O。

一个块中仅是一个关系中的元组，所以最多遍历 1000 次磁盘块就能找到所有元组。

- 如果  $R$  不是聚簇的，且不使用索引，查询代价 = 20000 个 I/O。

一个块中不仅是一个关系中的元组，可能含有其他数据，必须通过元组的指针进行遍历，所以要访问 20000 个元组，考虑最坏情况，相邻元组都不在同一个块，那么就要访问 20000 个块。

$V(A, R)$  表示  $R$  中属性  $A$  出现不同值的数目，即  $\Pi_A(R)$  的数目。

PPT 中  $A, R$  的顺序时常发生改变，询问老师说都可以。书上是属性在前，关系名在后。为了统一写法，下面全部采用书本的形式，因此和 PPT 几乎都是相反的。

- 如果  $V(a, R) = 100$  且索引是聚簇的，查询代价估计  $= 1000/100 = 10$  个 I/O。

有索引的帮助，可以快速定位到这个元组，再加上是聚簇的，所以相同值的也是相邻的。

因为有 100 个不同的值，所以在最坏情况下平均分散到 10 个块里面。

- 如果  $V(a, R) = 100$  且索引是非聚簇的，查询代价  $= 20000/100 = 200$  个 I/O。
- 如果  $V(a, R) = 20000$ ，即  $a$  是关键字，主键索引，查询代价  $= 20000/20000 = 1$  个 I/O，不管是否是聚簇的。

- 基于有序索引的连接算法 (Zig-Zag 连接算法) :

- 以  $R$  和  $S$  做等值连接为例。
- $R$  和  $S$  都有在  $Y$  属性上的 B+ 树索引。
- $R$  和  $S$  均从左至右读取索引树的叶子结点:

1. 读  $R$  的第一个索引项赋予  $a$ ，再读  $S$  的第一个索引项赋予  $b$ ;
2. 如果  $a \neq b$ ，则:
  - 如果  $a < b$ ，则将  $R$  的下一个索引项赋予  $a$ ，继续执行 2。
  - 如果  $a > b$ ，则将  $S$  的下一个索引项赋予  $b$ ，继续执行 2。
3. 如果  $a = b$ ，则将  $R$  和  $S$  关系中对应的元组读出并进行连接，直到  $R$  的所有相等的  $a$  值和  $S$  的所有相等的  $b$  值对应的元组都处理完毕，将  $R$  的下一个索引项赋予  $a$ ，将  $S$  的下一个索引项赋予  $b$ ，继续执行 2。

## 5.5. 两阶段多路归并排序 TPMMS

实验做，考试老师说不会考。

- 基本思路:

- 第一趟:
  - 划分子集，并使子集具有某种特性，如有序或相同散列值等。

- 第二趟:

处理全局性内容的操作，形成结果关系。如多子集间的归并排序，相同散列值子集的操作等。

- 内排序:

待排序的数据可一次性地装入内存中，即排序者可以完整地看到和操纵所有数据。内存中数据的排序算法有插入排序、选择排序、冒泡排序等。

- 外排序:

待排序的数据不能一次性装入内存，即排序者不能一次完整地看到和操纵所有数据，需要将数据分批装入内存分批处理的排序问题。

- 多路归并的过程 (仅简单分析) :

- 设内存块数为  $B_{memory}$ ，待排序数据块数为  $B_{problem}$ 。
- 第一趟划分子集并排序，这里要求每个子集的块数要小于等于内存块数  $B_{memory}$ ，I/O 次数为  $2B_{problem}$ 。
- 第二趟各子集间的归并排序，这里要求子集的数目要小于内存块数  $B_{memory}$ ，I/O 次数为  $2B_{problem}$ 。

这两个限制说明大数据集块数  $\leq B_{memory}^2$ 。

同时内存的块中得有一块用来输出（第一趟不需要输出块），一块用来比较（老师说可以没有）。

- 算法的效率:

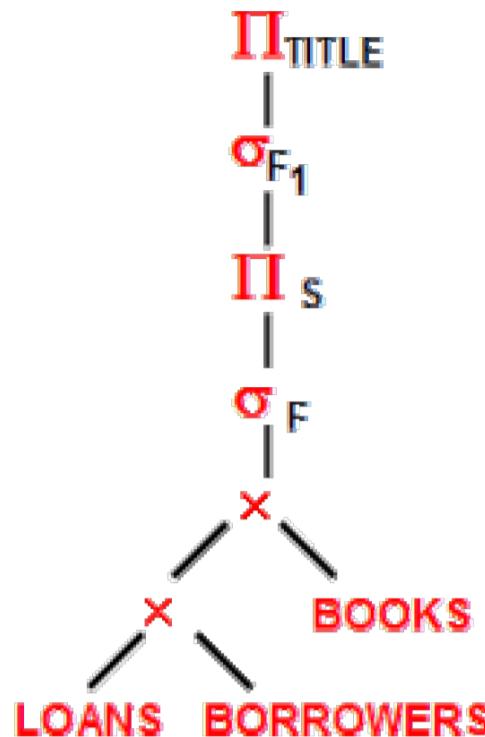
读写磁盘块的次数，即 I/O 数  $= 4B_{problem}$ 。

## 5.6. 语法优化技术

必考，估计选择填空题。

- 通过语法树，表达和分析关系代数表达式:

- 长这样：



从叶子到树根执行。

- 策略：

- 尽可能提前选择和投影（相当于这两个运算在语法树上尽量下放）：
  - 可使中间结果变小，节省几个数量级的执行时间。
- 选择与投影尽量一起做：
  - 投影或选择后，下一个投影或选择操作通过中间结果进行操作，这样就不需要再从磁盘读数据。
- 投影与其前后的二元运算结合：
  - 通过投影，去掉关系的无关属性（不参与二元运算的属性），可以避免多次扫描整个关系。

PPT 似乎只使用了以上三个策略，后面 MOOC 中有使用，还是按 PPT 来吧。

- 笛卡尔积转连接运算（把选择与其前的笛卡尔积合并成一个连接）：
  - 当  $R \times S$  前有选择运算且其中存在条件是  $R, S$  属性间比较的运算时，可将其转化为连接运算可节省时间。
- 执行连接运算前对关系做适当预处理：
  - 文件排序、建立临时索引等，可使两关系公共值高效联接。
- 找出表达式里的公共子表达式：
  - 若公共子表达式结果不大，则预先计算，以后可读入此结果，节时多，尤当视图情况下有用。

- 关系代数操作次序交换的等价性：

- $L_1$ ，连接与积（并交）的交换律：

设  $E_1, E_2$  是关系代数表达式， $F$  是  $E_1, E_2$  中属性的附加限制条件，则有：

$$\begin{aligned} E_1 \bowtie E_2 &\equiv E_2 \bowtie E_1 \\ E_1 \bowtie_F E_2 &\equiv E_2 \bowtie_F E_1 \\ E_1 \times E_2 &\equiv E_2 \times E_1 \end{aligned}$$

通过交换，可以将  $E_1, E_2, \dots, E_n$  中结果小集合小的放入内存，进行连接或者积操作，可以减小中间结果。

- $L_2$ ，连接与积（并交）的结合律：

若  $E_1, E_2, E_3$  是关系代数表达式， $F_1, F_2$  是条件，则有：

$$\begin{aligned} (E_1 \bowtie_{F_1} E_2) \bowtie_{F_2} E_3 &\equiv E_1 \bowtie_{F_1} (E_2 \bowtie_{F_2} E_3) \\ (E_1 \bowtie E_2) \bowtie E_3 &\equiv E_1 \bowtie (E_2 \bowtie E_3) \\ (E_1 \times E_2) \times E_3 &\equiv E_1 \times (E_2 \times E_3) \end{aligned}$$

结合律和交换律说明结果与运算顺序无关，可以将  $E_1, E_2, \dots, E_n$  中结果集合排序，升序放入内存，进行连接或者积操作，可以减小中间结果。

◦  $L_3$ , 投影串接律 (双向使用) :

设属性集  $\{A_1, \dots, A_n\} \subseteq \{B_1, \dots, B_m\}$ ,  $E$  是表达式, 则有:

$$\Pi_{A_1, \dots, A_n}(\Pi_{B_1, \dots, B_m}(E)) \equiv \Pi_{A_1, \dots, A_n}(E)$$

从左往右使用, 两遍扫描变为一遍扫描, 减少 IO 次数。

从右到左使用, 可以用于扩充, 便于将投影在语法树中下放, 减少无关属性。

◦  $L_4$ , 选择串接律 (双向使用) :

若  $E$  是关系代数表达式,  $F_1, F_2$  是条件, 则有:

$$\sigma_{F_1}(\sigma_{F_2}(E)) \equiv \sigma_{F_1 \wedge F_2}(E)$$

理由其实同上, 只是扩充的时候是选择下放。

◦  $L_5$ , 选择和投影的交换律:

- 设条件  $F$  只涉及属性  $\{A_1, \dots, A_n\}$ ,  $E$  是关系表达式, 则有:

$$\Pi_{A_1, \dots, A_n}(\sigma_F(E)) \equiv \sigma_F(\Pi_{A_1, \dots, A_n}(E))$$

- 更一般地, 若  $F$  还涉及不属于  $\{A_1, \dots, A_n\}$  的属性  $\{B_1, \dots, B_m\}$ , 则:

$$\Pi_{A_1, \dots, A_n}(\sigma_F(E)) \equiv \Pi_{A_1, \dots, A_n}(\sigma_F(\pi_{A_1, \dots, A_n, B_1, \dots, B_m}(E)))$$

◦  $L_6$ , 选择和积的交换律:

设  $E_1, E_2$  是关系代数表达式:

- 若条件  $F$  只涉及  $E_1$  中的属性, 则有:

$$\sigma_F(E_1 \times E_2) \equiv \sigma_F(E_1) \times E_2$$

- 若  $F = F_1 \wedge F_2$ ,  $F_1, F_2$  分别只涉及  $E_1, E_2$  中属性, 则有:

$$\sigma_F(E_1 \times E_2) \equiv \sigma_{F_1}(E_1) \times \sigma_{F_2}(E_2)$$

- 若  $F = F_1 \wedge F_2$ ,  $F_1$  只涉及  $E_1$  中属性, 而  $F_2$  涉及  $E_1, E_2$  中属性, 则有:

$$\sigma_F(E_1 \times E_2) \equiv \sigma_{F_2}(\sigma_{F_1}(E_1) \times E_2)$$

◦  $L_7$ , 投影和积的交换律:

设  $E_1, E_2$  为两关系代数表达式,  $A_1, \dots, A_n$  是出现在  $E_1$  或  $E_2$  中的一些属性, 其中  $B_1, \dots, B_m$  出现在  $E_1$  中, 剩余的属性  $C_1, \dots, C_k$  出现在  $E_2$  中, 则有:

$$\Pi_{A_1, \dots, A_n}(E_1 \times E_2) \equiv \Pi_{B_1, \dots, B_m}(E_1) \times \Pi_{C_1, \dots, C_k}(E_2)$$

◦  $L_8$ , 选择和并的交换律:

设关系代数表达式  $E = E_1 \cup E_2$ ,  $F$  是条件, 则有:

$$\sigma_F(E_1 \cup E_2) \equiv \sigma_F(E_1) \cup \sigma_F(E_2)$$

此定理要求  $E_1, E_2$  是并相容的。

◦  $L_9$ , 选择和差的交换律:

设关系代数表达式  $E = E_1 - E_2$ ,  $F$  是条件, 则有:

$$\sigma_F(E_1 - E_2) \equiv \sigma_F(E_1) - \sigma_F(E_2)$$

◦  $L_{10}$ , 投影和并的交换律:

设关系代数表达式  $E = E_1 \cup E_2$ ,  $A_1, \dots, A_n$  是  $E$  中的一些属性, 则有:

$$\Pi_{A_1, \dots, A_n}(E_1 \cup E_2) \equiv \Pi_{A_1, \dots, A_n}(E_1) \cup \pi_{A_1, \dots, A_n}(E_2)$$

因为投影会去重，[投影和差的交换律](#)是不成立的，即：

$$\Pi_{A_1, \dots, A_n}(E_1 - E_2) \not\equiv \Pi_{A_1, \dots, A_n}(E_1) - \pi_{A_1, \dots, A_n}(E_2)$$

一个大概的例子：

假设  $E_1$  只有 3 个元组，在属性  $A_1, \dots, A_n$  上都相等， $E_2$  只有一个元组，与  $E_1$  的 3 个元组之一相等。

- 先差后投影：

$E_1 - E_2$  有两个元组，去重后有 1 个元组。

- 先投影后差：

投影后  $E_1, E_2$  都只有一个元组，差运算后结果为空，与前者不等价。

- 优化算法执行流程：

- 依据选择串接律  $\sigma_{F_1}(\sigma_{F_2}(E)) \equiv \sigma_{F_1 \wedge F_2}(E)$ ，对于右边形式的关系表达式，转为左边串接形式。
- 对每个选择和投影，依据定理  $L_4$  至  $L_9$ ，尽可能把它下放（如果一个投影是对某表达式所有属性进行的，相当于 `select *`，则去掉）
- 依据选择串接律和投影串接律把串接的选择和投影组合为单个选择、单个投影。

笔者感觉以上修改要多次使用，直到不能再应用为止。

- 对修改后的语法树，将其内结点按以下方式[分组](#)：

- 每个二元运算结点（积、并、差、连接等）和其所有[一元运算直接祖先结点](#)放在一组
- 对于其后代结点，若后代结点是一串[一元运算且以树叶为终点](#)，则将这些一元运算结点放在该组中。
- 若后代节点有二元运算结点[笛卡儿积](#)，则不能将后代结点归入该组。
- 笔者认为可能是从叶子节点向上分组，这样遇到一个二元运算节点就做判断，这个二元节点向上的一元运算归为一组，向下到叶子节点的运算也归为这组，然后整个组就独立执行，输出一个结果集合，相当于一个叶子节点了。重复这个过程直到执行完毕。

- 语法树执行顺序为：

以每组结点为一步，后代先执行，从叶子执行到根。

## 5.7. 代价估计

- 统计信息：

- $T_R$  或  $T(R)$ ：

关系  $R$  的元组数目。

- $V(A, R)$ ：

$R$  中属性  $A$  出现不同值的数目，即  $\Pi_A(R)$  的数目。

PPT 中  $A, R$  的顺序时常发生改变，询问老师说都可以。书上是属性在前，关系名在后。为了统一写法，下面全部采用书本的形式，因此和 PPT 几乎都是相反的。

- 估计的目标：

给定一个表达式  $E$ ，如何估计  $E$  的元组数目  $T(E)$ 。

- 估算  $\pi_A(R)$  的大小：

PPT：

$$T(\pi_A(R)) = T(R)$$

笔者认为应该是：

$$T(\pi_A(R)) = V(A, R)$$

投影运算并未减少行数，但可能有效地减少了存储结果关系的[块数](#)（每个元组所占的大小减少）。

- 估算选择运算  $S = \sigma_{A=c}(R)$  的大小：

严谨:

$T(S)$  介于  $[0, T(R) - V(A, R) + 1]$  之间

原因分析:

- 最小值为 0，因为可能关系  $R$  中的  $A$  属性不存在值为  $c$  的元组。
- 最大值为  $T(R) - V(A, R) + 1$ ，即关系  $R$  中的  $A$  属性值不为  $c$  的元组数目都是 1，共有  $V(A, R)$ 。

估计:

$T(S) = T(R)/V(A, R)$ ，即  $A$  属性不同值的元组数相等。

暴力估计:

不知道  $V(R, A)$  时，默认为 10，即  $T(S) = T(R)/10$ 。

- 估算选择运算  $S = \sigma_{A < c}(R)$  的大小:

严谨:

$T(S)$  介于 0 to  $T(R)$  之间

分析:

- 最小值 0 即不存在这样的元组。
- 最多  $T(R)$  即所有元组都满足条件。

估计:

$T(S) = T(R)/2$ ，应有一半的元组满足条件。

实际常用估计:

$T(S) = T(R)/3$

- 估算选择运算  $S = \sigma_{A=10 \text{ and } B<20}(R)$  的大小

估计:

$T(S) = T(R)/(V(R, A)^*3)$

分析:

- $\sigma_{A=10 \text{ and } B<20}(R) = \sigma_{B<20}(\sigma_{A=10}(R))$
- $A = 10$ ，得出  $T(S) = T(R)/V(R, A)$
- $B < 20$ ，得出  $T(S) = T(S)/3$

- 估算选择运算  $S = \sigma_{C_1 \text{ or } C_2}(R)$  的大小:

估计:

$T(S) = n(1 - (1 - \frac{m_1}{n})(1 - \frac{m_2}{n}))$

分析:

- $R$  有  $n$  个元组，其中有  $m_1$  个满足  $C_1$ ，有  $m_2$  个满足  $C_2$ 。
- $(1 - \frac{m_1}{n})$  是不满足  $C_1$  的那些元组， $(1 - \frac{m_2}{n})$  是不满足  $C_2$  的那些元组。
- 两数之积是不满足条件的元组概率（这里并不严谨），1 减去这个积就是满足条件元组出现的概率。

- 前一种类型的举例，估计选择运算  $S = \sigma_{A=10 \text{ or } B<20}(R)$  的大小:

分析:

$$n = T(R) = 10000$$

$$V(R, A) = 50$$

$$m_1 = T(R)/V(R, A) = 10000/50 = 200$$

$$m_2 = T(R)/3 = 10000/3 \approx 3333$$

即有  $m_1$  个满足  $C_1$ , 有  $m_2$  个满足  $C_2$ ,  $(1 - \frac{m_1}{n})(1 - \frac{m_2}{n})$  是不满足这个条件的元组的概率, 计算如下:

$$T(S) = 10000 * (1 - (1 - 200/10000)(1 - 3333/10000)) \approx 3466$$

也可以简单估计为  $T(S) = T(R)/3 = 10000/3 \approx 3333$

- 估算连接运算  $S' = R(X, Y) \text{ Natural Join } S(Y, Z)$  的大小:

估计:

$$T(S') = \frac{T(R)T(S)}{\max(V(Y, R), V(Y, S))}$$

分析:

- 假定  $V(Y, R) \geq V(Y, S)$ ,  $R$  中元组  $r$  和  $S$  中元组有相同  $Y$  值的概率  $= 1/V(Y, R)$
- 假定  $V(Y, R) < V(Y, S)$ ,  $R$  中元组  $r$  和  $S$  中元组有相同  $Y$  值的概率  $= 1/V(Y, S)$
- 则笛卡尔积后的关系在  $Y$  上相等的概率  $= 1/\max(V(R, Y), V(S, Y))$
- 笔者尝试解释:
  - 首先假设数据是均匀分布, 即假设一个关系  $U$  的  $Y$  属性中, 具有相同属性值的元组数均为  $x$  个, 这样总元组数是  $x \times V(Y, U)$ 。
  - $R$  关系的总元组数为  $x_R \times V(Y, R)$ ,  $S$  关系的总元组数为  $x_S \times V(Y, S)$ 。
  - 假设  $R.Y \subseteq S.Y, V(Y, R) < V(Y, S)$ , 即  $R$  的  $Y$  属性值在  $S$  中均存在。这使得  $R$  的每个元组, 都能等值连接  $x_S$  个  $S$  的元组。
  - 所以  $R \times S$  中, 满足  $R.Y = S.Y$  的元组数为  $x_R \times V(Y, R) \times x_S$ 。然后除以总元组数即可得到概率 (其实分子就是估计值  $T(S')$ ) :

$$\frac{x_R \times V(Y, R) \times x_S}{x_R \times V(Y, R) \times x_S \times V(Y, S)} = \frac{1}{V(Y, S)}$$

- PPT 上的例子  $T(R) = 10000, T(S) = 50000, V(R, Y) = 500, V(S, Y) = 1000$ ,

估计为  $T(S') = 10000 * 50000 / 1000 = 500000$

## 6. 数据库事务处理技术

### 6.1. 事务 (纯复制)

- 事务的定义:
  - 事务是数据库管理系统提供的控制数据操作的一种手段。
  - 通过这一手段, 应用程序员将一系列的数据操作组合在一起作为一个整体进行操作和控制, 以便数据库管理系统能够提供一致性状态转换的保证。
- 事务的宏观性 (应用程序员看到的事务):

一个存取或改变数据库内容的程序的一次执行, 或者说一条或多条 SQL 语句的一次执行被看作一个事务。

事务一般是由应用程序员提出, 因此有开始和结束, 结束前需要提交或撤消 (通过 commit 或 rollback 确认的)。

- 事务的微观性 (DBMS 看到的事务):

对数据库的一系列基本操作 (读、写) 的一个整体性执行。

- 事务的并发执行:

多个事务从宏观上看是并行执行的, 但其微观上的基本操作 (读、写) 则可以是交叉执行的。

并发控制就是通过事务微观交错执行次序的正确安排, 保证事务宏观的独立性、完整性和正确性。

- 事务的特性 (ACID 特性):

- 原子性 (atomicity):

事务的所有操作在数据库中要么全部正确反映出来, 要么完全不反映。

- 一致性 (consistency) :

保证事务的操作状态是正确的，不能出现「丢失修改，不可重复读，脏读」三类错误，由隔离性保证。

- 丢失修改:

操作顺序是  $T_1$  读,  $T_2$  读,  $T_1$  写,  $T_2$  写。 $T_1$  的修改就被  $T_2$  所覆盖了，即丢失了修改。

- 不可重复读（理应能够重复读，但是现在不能重复读了，同一事务两次读期间未作修改，但读到的内容却不同）：

操作顺序是  $T_1$  读,  $T_2$  读,  $T_2$  写,  $T_1$  再读，发现与第一次读的数据不一致。

- 脏读:

操作顺序是  $T_2$  读,  $T_2$  写,  $T_1$  读,  $T_2$  回滚，此时  $T_1$  读到的是其他事务未提交的数据，回滚后这个数据已经无效了，称为脏数据。

- 隔离性 (isolation) :

尽管多个事务可能并发执行，但系统保证，对于任何一对事务  $T_i$  和  $T_j$ ，在  $T_i$  看来， $T_j$  或者在  $T_i$  开始之前已经完成执行，或者在  $T_i$  完成之后开始执行。因此，每个事务都感觉不到系统中有其他事务在并发地执行。

- 持久性 (durability) :

一个事务成功完成后，它对数据库的改变必须是永久的，即使出现系统故障。

可以说具有 ACID 特性的若干数据库基本操作的组合体被称为事务。

## 6.2. 事务调度

- 事务调度概念:

一组事务的基本步骤（读、写、其他控制操作如加锁、解锁等）的一种执行顺序称为对这组事务的一个调度。

- 并发调度的正确性:

并发调度下所得到的新数据库结果与分别串行地运行这些事务所得的新数据库完全一致，调度才是正确的，是结果意义上的正确。

- 可串行性:

如果不管数据库初始状态如何，一个调度对数据库状态的影响都和某个串行调度相同，则我们说这个调度是可串行化的或具有可串行性。

可串行化调度一定是正确的并行调度，但正确的并行调度，却未必都是可串行化的调度。

因为并行调度的正确性是指内容上结果正确性（结果对就对了），而可串行性是指形式上结果正确性，便于操作。

- 一种简单的事务调度的标记模型:

- $r_T(A)$ :

事务  $T$  读  $A$ 。

- $w_T(A)$ :

事务  $T$  写  $A$ 。

- 冲突:

◦ 如果调度中两个动作的顺序交换，若涉及的事务中至少有一个事务的行为会改变，则称这两个动作冲突。

◦ 有冲突的两个操作是不能交换次序的，没有冲突的两个事务是可交换的

◦ 几种冲突的情况:

- 同一事务的任何两个操作都是冲突的:

$$r_i(X); w_i(Y) \quad w_i(X); r_i(Y)$$

- 不同事务对同一元素的两个写操作是冲突的:

$$w_i(X); w_j(X)$$

- 不同事务对同一元素的一读一写操作是冲突的:

$$w_i(X); r_j(X) \quad r_i(X); w_j(X)$$

- 冲突可串行性:

一个调度，如果通过交换相邻两个无冲突的操作能够转换到某一个串行的调度（注意不是可串行），则称此调度为冲突可串行化的调度。

冲突可串行性是比可串行性更严格的概念，即冲突可串行性是可串行性的充分不必要条件。

例子：

$w_1(Y); w_2(Y); w_2(X); w_1(X); w_3(X)$  是一个可串行化的调度，因为它与串行调度  $w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X)$  的结果等价。

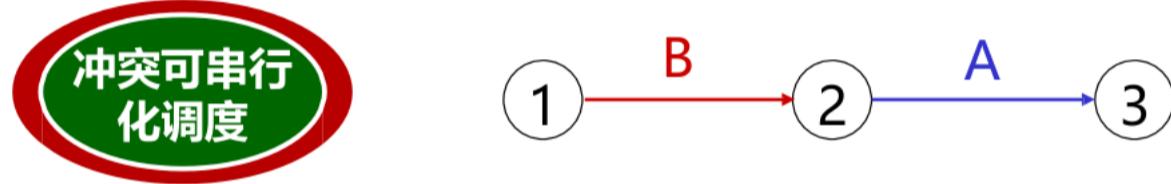
但是它相邻动作都是冲突的，不能交换形成串行调度，所以不是冲突可串行性的调度。

- 冲突可串行性判别算法：

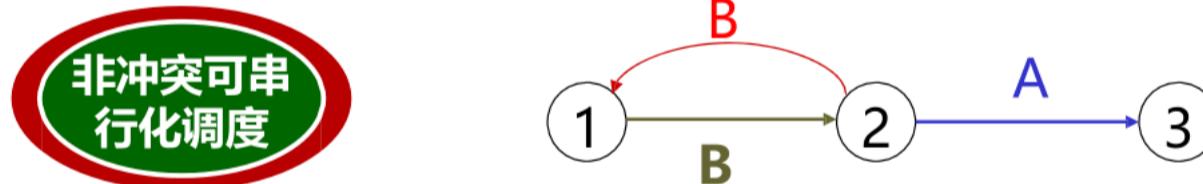
- 构造一个有向图，结点是每一个事务  $T_i$ 。
- 如果  $T_i$  的一个操作与  $T_j$  的一个操作发生冲突，且  $T_i$  在  $T_j$  前执行，则绘制一条由  $T_j$  指向  $T_i$  的有向边，表示  $T_i$  要在  $T_j$  前执行。
- 如果此有向图没有环，则是冲突可串行化的。

例子：

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



与边上的元素无关，只要存在环就是不满足冲突可串行的。

## 6.3. 锁

- 锁的类型：

- 排他锁 X：

只有一个事务能读、写，其他任何事务都不能读、写。

- 共享锁 S：

所有事务都可以读，但任何事务都不能写。

- 更新锁 U：

初始读，以后可升级为写。

- 增量锁 I（后面不涉及）：

增量更新（例如  $A = A + x$ ）区分增量更新和其他类型的更新。

- 相容性矩阵：

- 当某事务对一数据对象持有一种锁时，另一事务再申请对该对象加某一类型的锁，是允许（填「是」）还是不允许（填「否」）。
- 有更新锁的情况：

		申请锁	申请锁	申请锁
		共享锁 S	排他锁 X	更新锁 U
持有锁的模式	共享锁 S	是	否	是
持有锁的模式	排他锁 X	否	否	否
持有锁的模式	更新锁 U	否	否	否

- 锁协议:

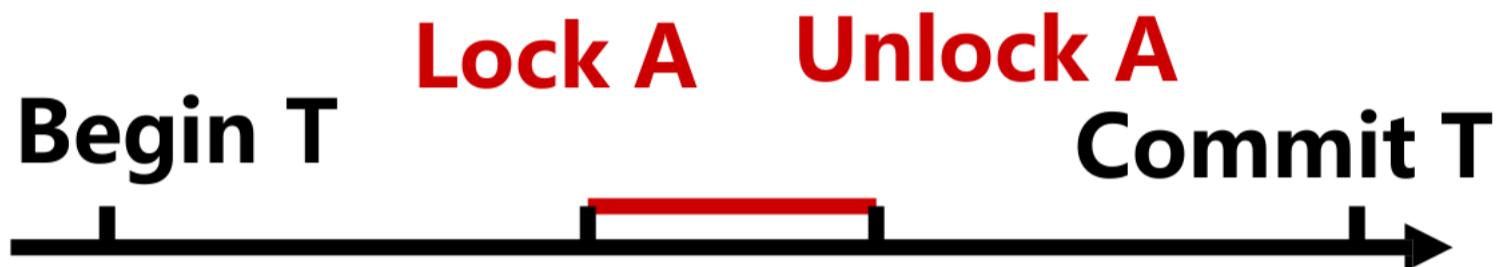
重难点，PPT 更改了很多次，也特意问了老师和同学，希望下面理解是正确的。

- 背景:

一开始并发控制的时候，读写完全没有锁，如果不是可串行化调度就会出错。

- 0 级协议:

- 所有写操作都要加排他锁，但排他锁是写完就释放，而不是直到提交事务才释放。

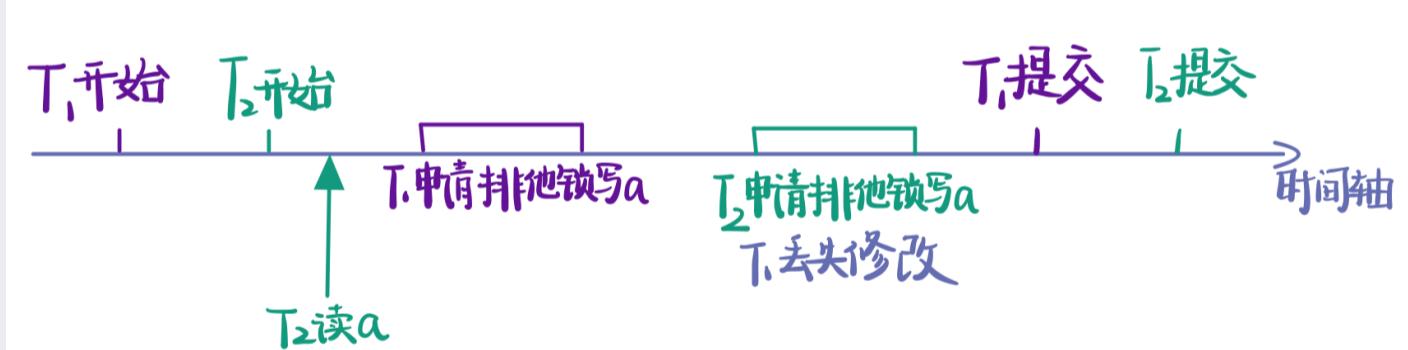


注意在排他锁期间是可以读的，读操作并没有要求上锁，也就不会因为申请有关读的锁而阻塞。

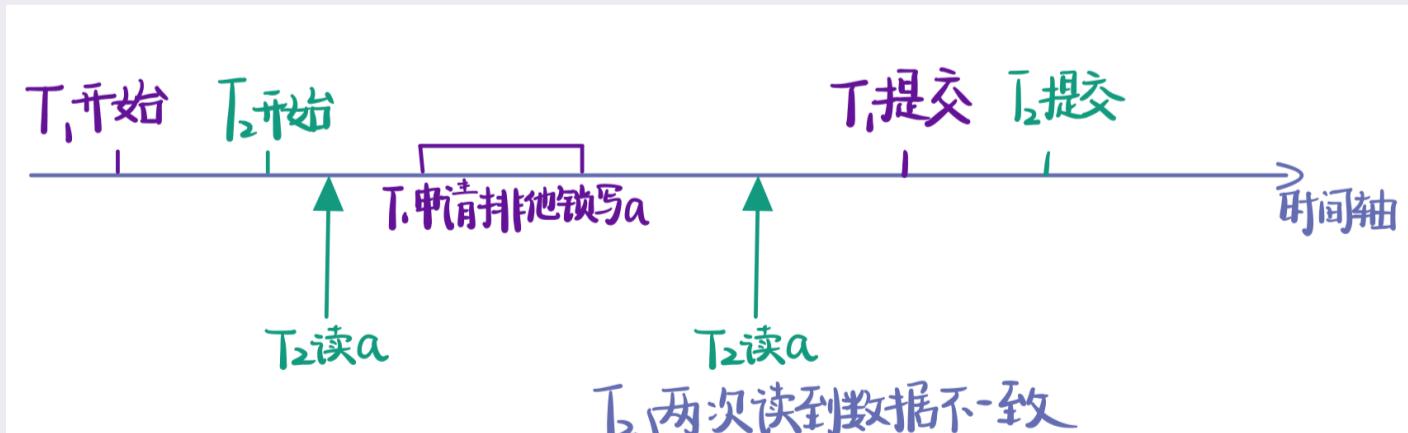
- 不能防止丢失修改，不可重复读，脏读三种不一致错误，完全没用属于是。

分析:

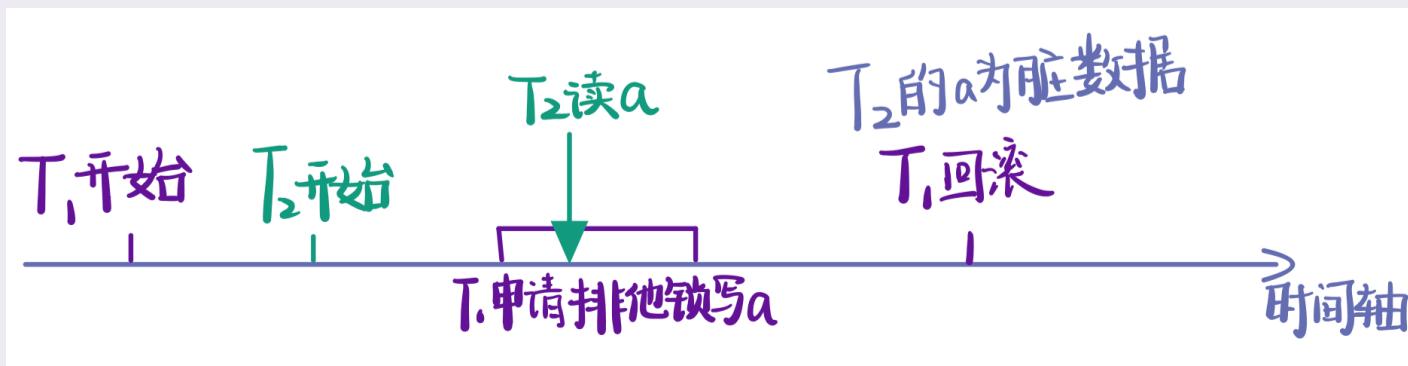
- 丢失修改:



- 不可重复读:



- 脏读:



- 1 级协议:

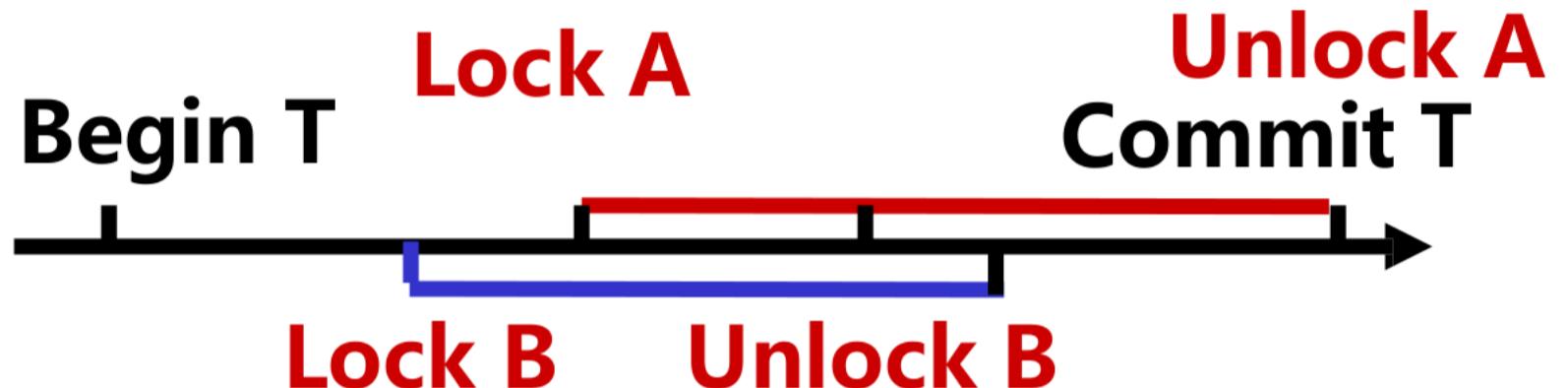
- 所有写操作都要加排他锁，排他锁直到提交事务才释放。



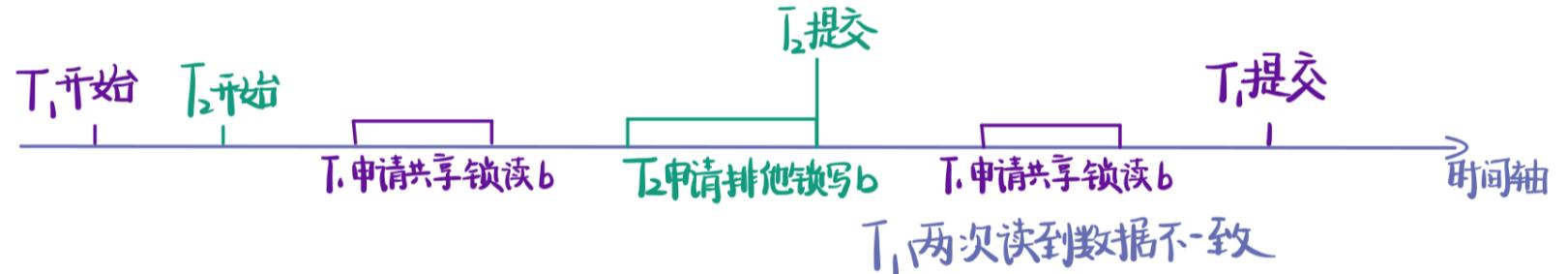
- 可以防止丢失修改，因为在别人不能在你提交前进行修改（有写排他锁），这样你的修改将可以一直保留到提交。
- 不能防止不可重复读和脏读，理由完全同 0 级协议的图，因为对读没有任何限制，你排他锁关我读操作什么事。

- 2 级协议:

- 所有写操作同 1 级协议，读操作要加共享锁，但共享锁是读完就释放，而不是直到提交事务才释放。

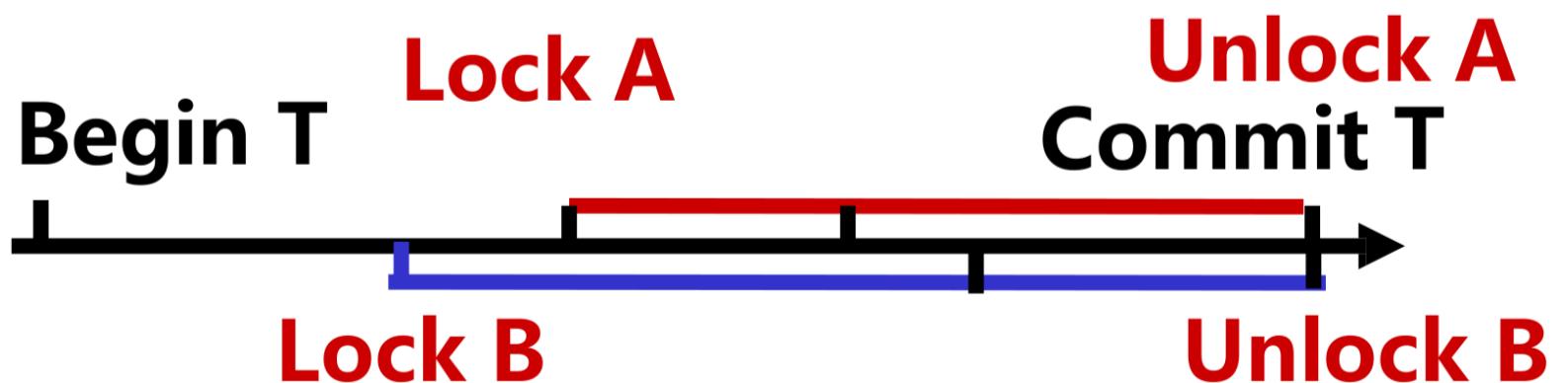


- 可以防止丢失修改，理由同 1 级协议。
- 可以防止脏读，因为读操作需要申请共享锁，你能申请到共享锁，是因为这个元素没有被上排他锁，所以就不会有对该元素的写操作，那么即使有回滚也和该元素无关。
- 不能防止不可重复读，理由见下图:



- 3 级协议:

- 所有写操作同 1 级协议，读操作要加共享锁，直到提交事务才释放。



- 防止丢失修改，不可重复读，脏读。

不可重复读现象不会发生，因为不会出现上图的共享锁释放后还可以被别人修改，且我再读的现象。

- 封锁粒度:

- 定义:

封锁粒度是指封锁数据对象的大小。

- 粒度单位:

属性值 < 元组 < 元组集合 < 整个关系 < 整个 DB

- 由前往后：

并发度小，封锁开销小。

- 两段封锁协议：

- 每个事务分两个阶段提出加锁和解锁申请。

1. 增长阶段：

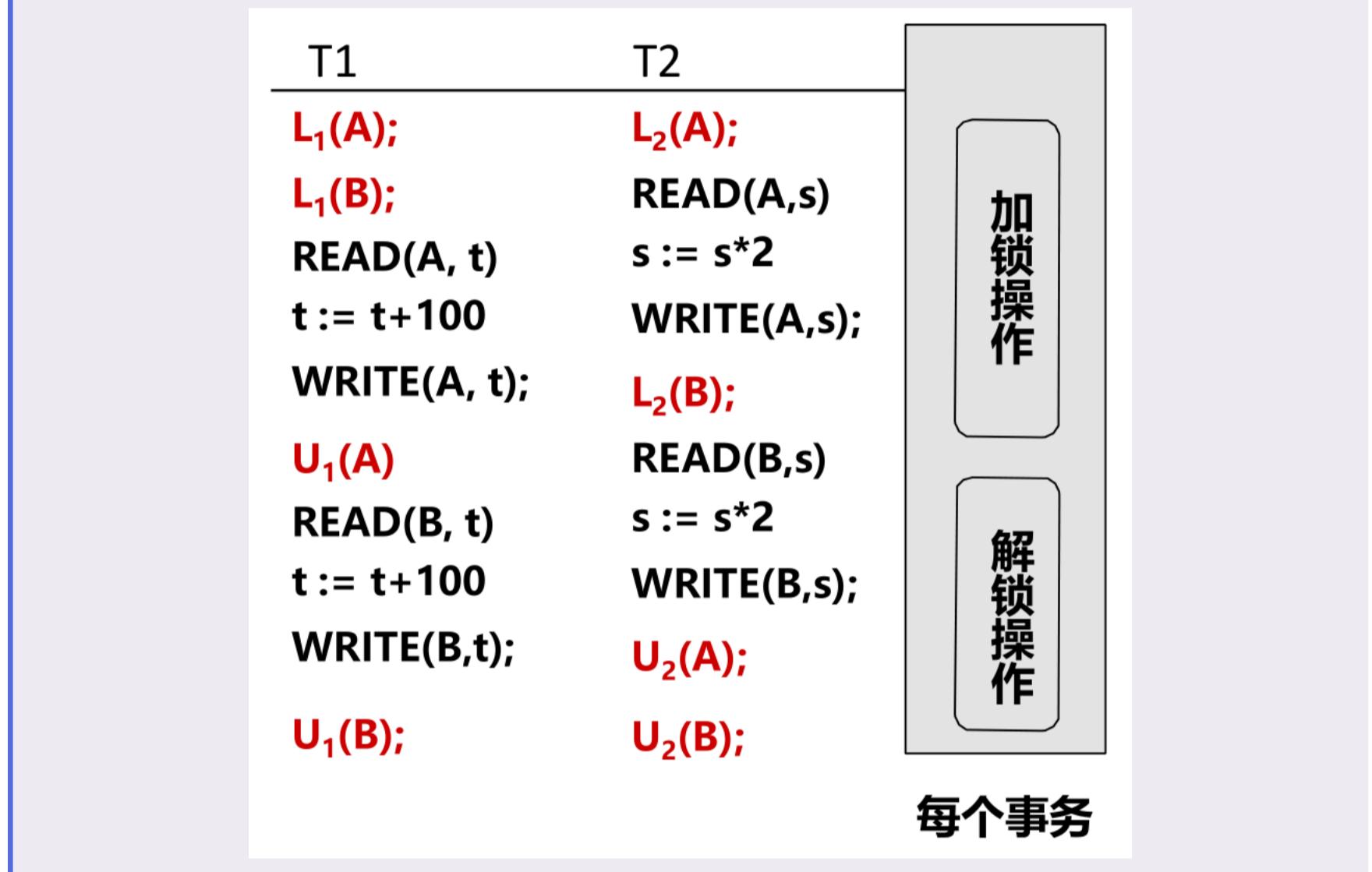
事务可以获得锁，但不能释放锁。

2. 缩减阶段：

事务可以释放锁，但不能获得新锁。

等价于 PPT 中的加锁段中不能有解锁操作，解锁段中不能有加锁操作。

每个事务中所有封锁请求先于任何一个解锁请求。



- 可以保证冲突可串行性的，但是可能产生死锁。

## 6.4. 基于时间戳的并发控制方法

- 不用锁实现并发控制。

- 时间戳定义：

时间戳是一种基于时间的标志，将某一时刻转换成的一个数值，具有唯一性和递增性。

- 基本思想：

- 事务  $T$  启动时，系统将该时刻作为  $T$  的时间戳。
- 时间戳可以表征一系列事务执行的先后次序，时间戳小的事务先执行，时间戳大的事务后执行（但是实际上是交叉执行的）。
- 强制事务调度等价于一个特定顺序（即时间戳升序）的串行调度。
- 对于每个动作都要判断是否存在时间戳上的冲突（保证时间戳小的先操作，大的后操作）：
  - 如无冲突，予以执行；
  - 如有冲突，则撤销这个动作对应的事务，并重启该事务，此时该事务获得了一个更大的时间戳，对应的动作位置会发生改变，重新做冲突判断。

回滚 = 撤销 + 重启

- 冲突类型:

1. 时间戳大的先写（读），时间戳小的后读（写）
2. 时间戳大的先写，时间戳小的后写

- 简单调度:

- 对 DB 中的每个数据元素  $x$ ，系统保留其上的最大时间戳:

- $RT(x)$ :

读过  $x$  的事务中最大的时间戳。

- $WT(x)$ :

写过  $x$  的事务中最大的时间戳。

- 事务的时间戳  $TS(T)$

- 读 - 写并发（对应解决冲突类型 1）：

- 若  $T$  事务读  $x$ ，则比较  $TS(T)$  与  $WT(x)$ :

- 若  $TS(T) \geq WT(x)$  ( $T$  后进行)，则允许  $T$  操作，更改  $RT(x)$  为  $\max\{RT(x), TS(T)\}$
- 否则有冲突， $T$  回滚。

- 若  $T$  事务写  $x$ ，则比较  $TS(T)$  与  $RT(x)$ ，其实同时要考虑「写 - 写」并发:

- 若  $TS(T) \geq RT(x)$  ( $T$  后进行)，则允许  $T$  操作，更改  $WT(x)$  为  $TS(T)$

这里与 PPT 不同，笔者认为  $TS(T)$  不会比  $WT(x)$  小（即前者一定更大），否则会发生写 - 写冲突。

- 否则有冲突， $T$  回滚。

- 写 - 写并发（对应解决冲突类型 2）：

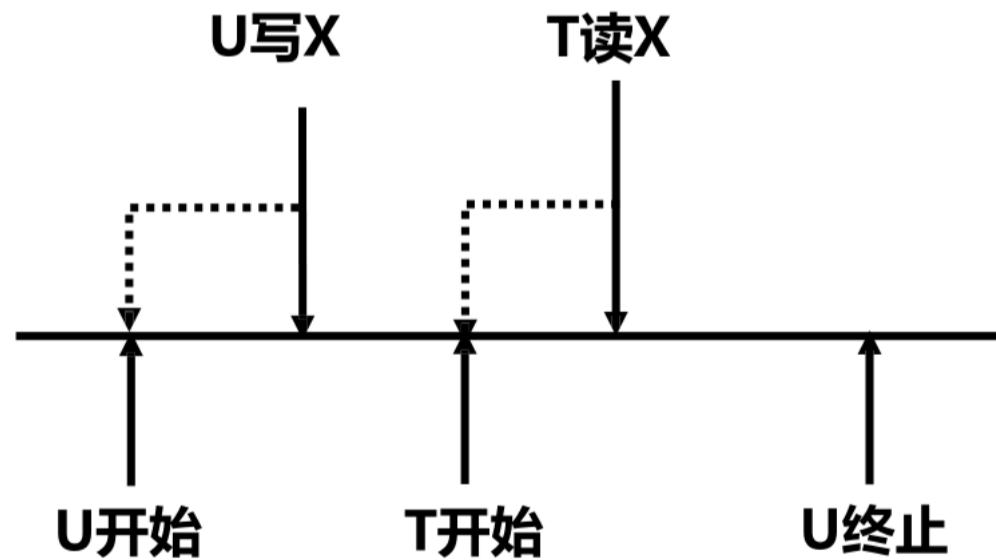
- 若  $T$  事务写  $x$ ，则比较  $TS(T)$  与  $WT(x)$ :

- 若  $TS(T) \geq WT(x)$ ，则允许  $T$  写，并且更改  $WT(x)$  为  $TS(T)$ ；

- 否则有冲突， $T$  回滚（或者采用所谓的托马斯写规则，直接忽略这个写而不是回滚整个事务，反正它已经被时间戳大的写覆盖了）。

- 以上简单调度会产生两种不一致错误（即时间戳并没有对这两种错误加以限制，只是消除了不可重复读错误）：

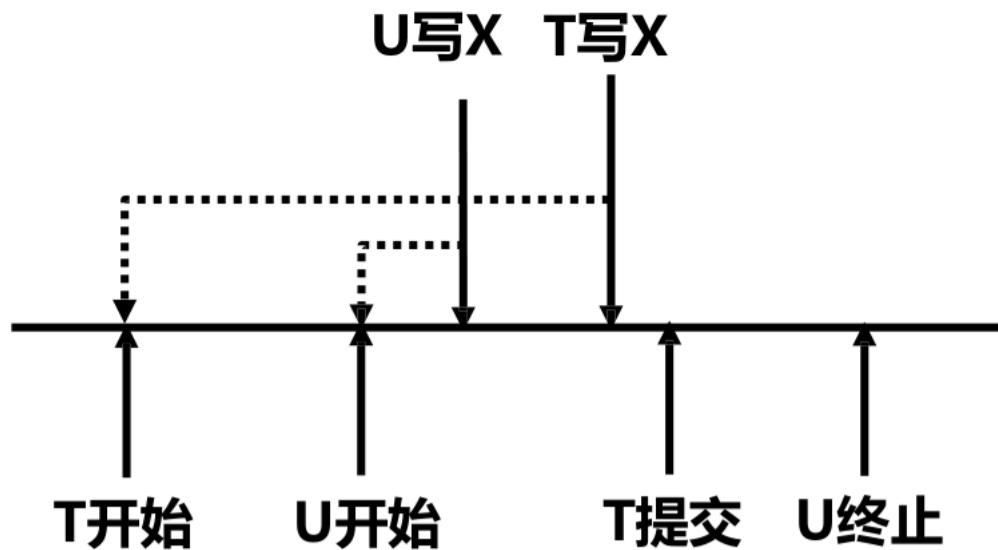
- 脏读:



## T读脏数据

事务  $T$  读取了事务  $U$  没提交的数据，在  $U$  回滚后就变成了脏数据。

- 类似丢失修改的错误:



## T由于U的写入而被跳过(无需写), 但U却终止了

$T$  的写在托马斯写规则下, 会被忽略, 但是  $U$  终止了, 导致谁都没用写成  $x$ 。这种情况下理应保留  $T$  的写。

- 优化后的时间调度:

- $WT(x), RT(x), TS(T)$  与简单调度定义相同。
- 增加提交位  $C(x)$ :
  - $C(x) = 1$  表明最近写  $x$  的事务 (时间戳较大) 已经提交, 就不会发生回滚现象。
  - $C(x)$  的意义: 将一些可能导致脏读和丢失修改的操作, 延迟到确定提交 (或回滚) 后再处理。
- 对来自事务  $T$  的读写请求, 调度器可以:
  - 同意请求
  - 回滚
  - 推迟, 并在以后决定是回滚还是要重新判断 (简单调度无此项选择)
- 若  $T$  事务读  $x$ , 则比较  $TS(T)$  与  $WT(x)$ :
  - 若  $TS(T) \geq WT(x)$  ( $T$  后进行), 理论上应该允许  $T$  操作, 但要判断最近的写是否提交:
    - $C(x) = 1$ , 说明已提交, 不会回滚, 同意请求, 可以放心的读, 不会是脏数据。更改  $RT(x)$  为  $\max\{RT(x), TS(T)\}$
    - 推迟  $T$  直到  $C(x) = 1$  或写  $x$  的事务终止

前一种情况仍可以放心读, 后一种情况就不能读了, 需要回滚。

- 否则有冲突,  $T$  回滚。
- 若  $T$  事务写  $x$ , 则比较  $TS(T), RT(x), WT(x)$ , 即同时要考虑「写 - 读」和「写 - 写」并发:
  - 若  $TS(T) \geq RT(x)$  且  $TS(T) \geq WT(x)$ , 则允许  $T$  操作, 更改  $WT(x)$  为  $TS(T)$ 。
  - 如果  $TS(T) \geq RT(x)$ , 但是  $TS(T) < WT(x)$ , 就要判断是否已提交
    - $C(x) = 1$ , 那么前一个  $x$  的写已提交, 则忽略  $T$  的写 (托马斯写规则)。
    - 否则推迟  $T$  直到  $C(x) = 1$  或写  $x$  的事务终止。

如果是前一种情况就会忽略  $T$  的写。后一种情况比较复杂, 我觉得不能简单地通过  $T$  的写请求:

- 假设现在有事务  $S_1, S_2, T$ , 时间戳排序为  $T < S_1 < S_2$ , 写  $x$  的顺序为  $S_1, S_2, T$ 。此时  $WT(x) = TS(S_2)$ , 且  $S_1$  已提交,  $S_2$  未提交。
  - $T$  的写请求会进入推迟判断, 即使最后  $S_2$  回滚了, 但是由于  $S_1$  的写已经提交, 也应该忽略  $T$  的写。
  - PPT 对这种情况没有处理办法 (如何感知  $S_1$  的存在), 所以这只是笔者的一种疑惑。

- 否则有冲突,  $T$  回滚。
- 假设调度器收到提交  $T$  的请求:
  - 它必须找到  $T$  所写的所有数据库元素  $x$ , 并置  $C(x) = 1$ 。
  - 等待  $x$  被提交的读动作可以继续读, 写动作要被忽略。
- 假设调度器收到终止  $T$  的请求:

任何等待  $T$  所写元素  $x$  的事务判断这一动作在  $T$  的写被终止后是否合法。

具体怎么判断就像上面说的一样还缺少一些信息作为判断依据。

## 6.5. 基于有效性确认的并发控制方法

- 基本思想:

- 事务在启动时刻被赋予唯一的时间戳，以示其启动顺序。

- $RS(T)$ :

- 事务  $T$  读数据的集合。

- $WS(T)$ :

- 事务  $T$  写数据的集合。

- 事务分三个阶段进行:

- 读阶段:

- 事务从数据库中读取读集合中的所有元素，在其局部地址空间计算它将要写的所有值。

- 有效性确认阶段:

- 调度器通过比较该事务与其它事务的读写集合来确认该事务的有效性。

- 写阶段:

- 通过有效性确认后，该事务立刻往数据库中写入其写集合中元素的值（这个过程要判断「读 - 写」和「写 - 写」冲突）。

- 并发事务串行的顺序即事务有效性确认的顺序。

- 集合定义:

- $START$  集合:

- 已经开始但尚未完成有效性确认的事务集合。

- 对此集合中的事务，调度器维护  $START(T)$ ，即  $T$  开始的时间。

- $VAL$  集合:

- 已经确认有效性但尚未完成写阶段写的事务。

- 对此集合中的事务，调度器维护  $START(T)$  和  $VAL(T)$ ，即  $T$  确认的时间。

- $FIN$  集合:

- 已经完成写阶段的事务。

- 对此集合中的事务，调度器记录  $START(T)$ ,  $VAL(T)$  和  $FIN(T)$ ，即  $T$  完成的时间。

- 有效性确认规则

- 读 - 写并发:

- 对于所有已经过有效性确认，且在  $T$  开始前没有完成的  $U$ ，即对于满足  $FIN(U) > START(T)$  的  $U$ ，检测  $RS(T) \cap WS(U)$  是否为空:

- 若为空，则确认  $T$ 。

- 否则，不予确认  $T$ 。

- 写 - 写并发:

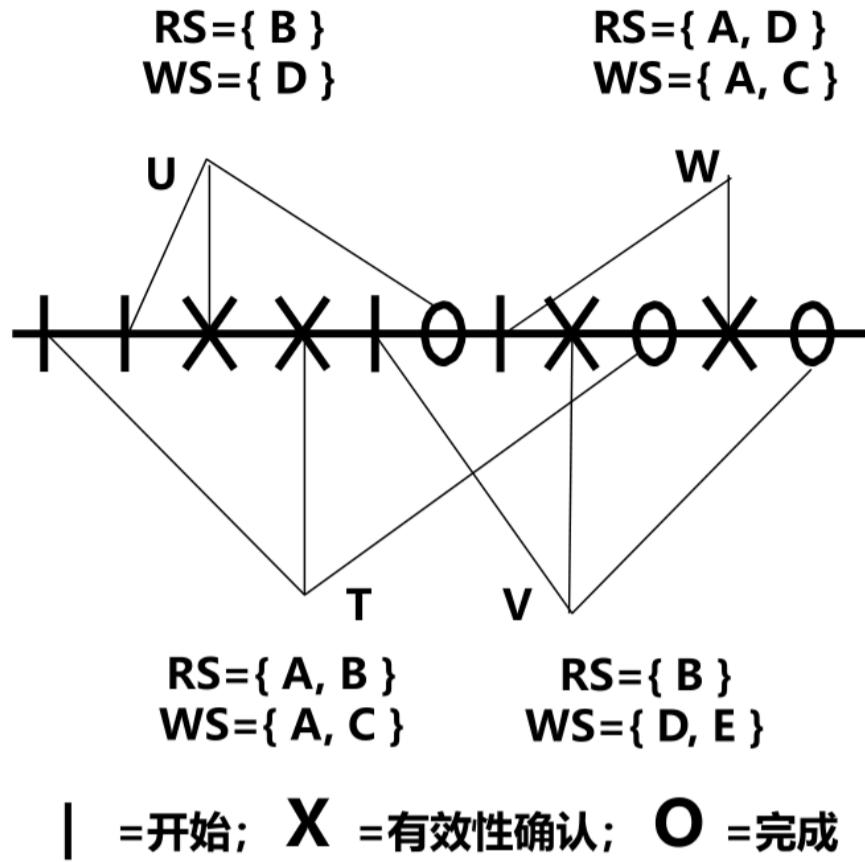
- 对于所有已经过有效性确认，且在  $T$  有效性确认前没有完成的  $U$ ，即对于满足  $FIN(U) > VAL(T)$  的  $U$ ，检测  $WS(T) \cap WS(U)$  是否为空:

- 若为空，则确认  $T$ 。

- 否则，不予确认  $T$ 。

- 例子:

## 示例：确认下列四个事务的有效性



### 1. U的有效性确认

无需检测，直接确认U。

### 2. T的有效性确认

因 $\text{FIN}(U) > \text{START}(T)$ , 需检测 $\text{RS}(T) \cap \text{WS}(U)$

因 $\text{FIN}(U) > \text{VAL}(T)$ , 需检测 $\text{WS}(T) \cap \text{WS}(U)$

检测结果：均为空，则确认T。

### 3. V的有效性确认

因 $\text{FIN}(U) > \text{START}(V)$ , 需检测 $\text{RS}(V) \cap \text{WS}(U)$

因 $\text{FIN}(T) > \text{START}(V)$ , 需检测 $\text{RS}(V) \cap \text{WS}(T)$

因 $\text{FIN}(T) > \text{VAL}(V)$ , 需检测 $\text{WS}(T) \cap \text{WS}(V)$

检测结果：均为空，则确认V。

### 4. W的有效性确认

因 $\text{FIN}(T) > \text{START}(W)$ , 需检测 $\text{RS}(W) \cap \text{WS}(T)$

因 $\text{FIN}(V) > \text{START}(W)$ , 需检测 $\text{RS}(W) \cap \text{WS}(V)$

因 $\text{FIN}(V) > \text{VAL}(W)$ , 需检测 $\text{WS}(V) \cap \text{WS}(W)$

检测结果：不全为空，则W不能确认，W被回滚。

## 7. 故障恢复

### 7.1. 故障恢复的宏观思路

- 故障类型：
  - 事务故障：  
某一个程序（事务）自身运行错误所引起的故障，影响该程序（事务）本身，只需要撤销事务和重做事务来进行恢复。
  - 系统故障：  
由于掉电、非正常关机等所引起的故障。影响正在运行的事务以及数据库缓冲区（数据库缓冲区涉及正在运行和已经运行的事务）。
  - 介质故障：  
由于介质（数据库）损坏等所引起的故障。影响是全面的，既影响内存中的数据，又影响介质中存储的数据。
- 故障会破坏事务的原子性，一致性和持久性。

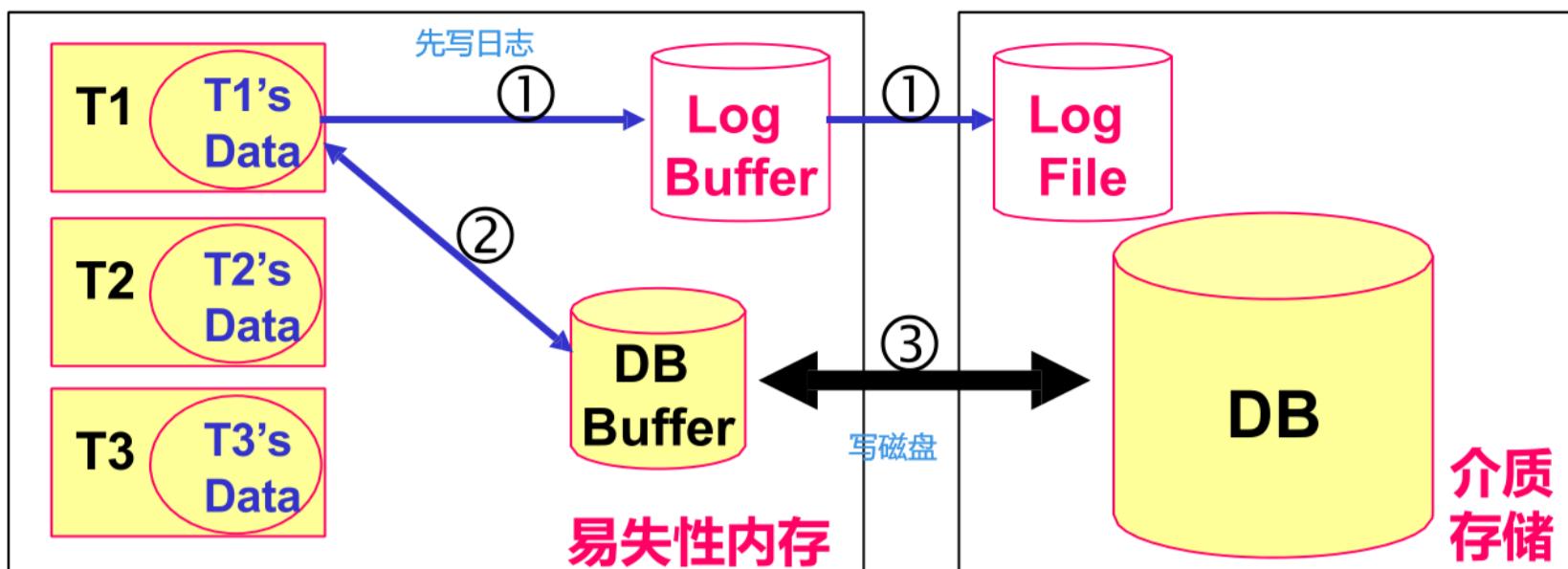
隔离性因为故障发生后不能执行事务，事务肯定是越少越不容易破坏隔离性。

- 故障恢复意义：  
把 DB 由当前不正确状态恢复到已知正确的某一状态。
- 需要保证事务的：
  - 原子性：  
事务的所有操作，要么全都执行，要么全都不执行。
  - 持久性：  
已提交的事务对数据库产生的影响是持久的（缓冲区内容保证写回磁盘），未提交的事务对数据库不应有影响（缓冲区内容不能影响磁盘）。
- 运行日志：
  - 运行日志是 DBMS 维护的一个文件，该文件以流水方式记录了每一个事务对数据库的每一次操作及操作顺序。

MOOC 上题目说「日志文件是用于记录对数据的所有更新操作」。

- 运行日志直接写入介质存储上，会保持正确性。

- 当事务对数据库进行操作时，先写运行日志。写成功后，再与数据库缓冲区进行信息交换。



- 保存信息：

- < Start T >, 表示事务 T 已经开始
- < Commit T >, 表示事务 T 成功完成
- < Abort T >, 事务 T 未成功, 被中止
- < T, X, v<sub>1</sub> > 或者 < T, X, v<sub>2</sub> > 或者 < T, X, v<sub>1</sub>, v<sub>2</sub> > (对应 Undo 型日志, Redo 型日志, Undo/Redo 型日志)
- 表示事务 T 改变了数据库元素 X, X 原来的值为 v<sub>1</sub> (X 的旧值) , X 新的值为 v<sub>2</sub>。

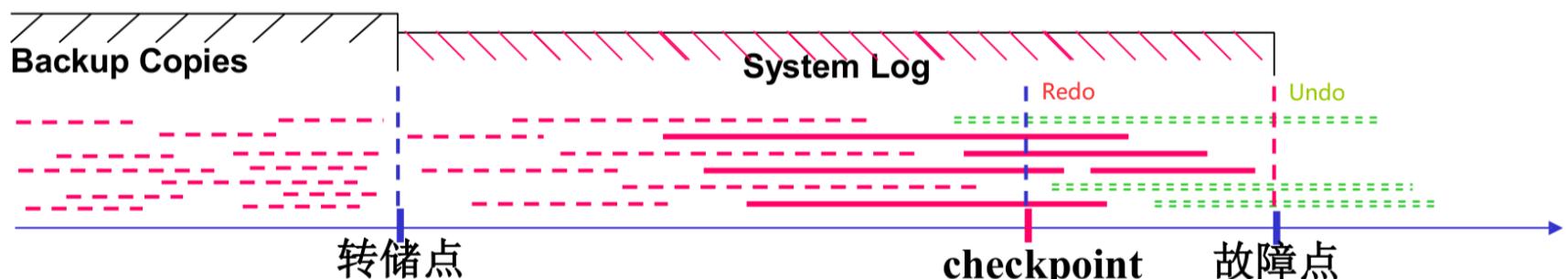
- 事务故障的恢复：

- 事务故障可通过重做事务 (Redo) 和撤消事务 (Undo) 来恢复。
- 重做事务可保证已提交事务的持久性，而撤销事务则消除未提交事务的影响。

已提交事务的写数据，可能只是放在缓冲区中，还没有写入磁盘，当发生故障时将丢失修改。所以必须通过 Redo 重新把修改写磁盘。

- DBMS 在运行日志中定期设置及更新检查点 (checkpoint) :

- 在检查点时，DBMS 强制使 DB 缓冲区中的内容与 DB 中的内容保持一致，即将 DB 缓冲区更新的所有内容写回 DB 中。
- 保证在检查点之前内存中数据与 DB 中数据是保持一致的，即检查点之前结束的事务不需要恢复（已经写回 DB）。
- 检查点之后结束或发生的事务需要依据运行日志进行恢复（不能确定是否写回 DB）：故障点前结束的重做，故障点时刻未结束的撤销。



- 通过副本实现介质故障恢复：

- 在某一时刻（转储点），对数据库在其他介质存储上产生的另一份等同记录，即副本。
- 当 DB 发生故障时用副本替换，且由于介质故障影响很大，替换后还需要依据运行日志进行恢复。
- 转储点的设置：
  - 过频会影响系统工作效率。
  - 过疏会造成运行日志过大，也影响系统运行性能（转储点前的日志不需要保存）。
  - 备份转储周期与运行日志的大小密切相关，合理设置。

## 7.2. 缓冲区

- 事务和缓冲区的操作：

- READ(X,t):

将元素 X 读到事务的局部变量 t 中。

- WRITE(X,t):  
将事务局部变量 t 写回元素 X。
  - INPUT(X):  
将元素 X 从磁盘读入到内存缓冲区中。
  - OUTPUT(X):  
将元素 X 从缓冲区写回到磁盘中。
  - COMMIT:  
事务提交。
  - ABORT:  
事务撤销。
- 缓冲区策略 :

- Force:  
缓冲区中的数据最晚在 commit 的时候写入磁盘。
- No Force:  
缓冲区中的数据可以一直保留，在 commit 之后过一段时间再写入磁盘。

**如果在系统崩溃的时候还没写入到磁盘，需要 Redo。**

- Steal:  
允许在事务 commit 之前把缓冲区中的数据写入磁盘（先偷偷写一点）。

**此时若系统在 commit 之前崩溃时，已经有数据写入到磁盘了，要恢复到崩溃前的状态，需要 Undo。**

- No Steal:  
不允许在事务 commit 之前把缓冲区中的数据写入磁盘。

**Steal/No Steal** 关心的缓冲区的数据最早什么时候开始写磁盘。  
**Force/No Force** 关心的缓冲区的数据最晚什么时候要写回磁盘。

- 各种搭配的效率和策略比较：

最灵活且常用 Steal + No Force

	No Steal	Steal
No Force		<b>最快</b>
Force	<b>最慢</b>	

**读写性能**

	No Steal	Steal
No Force	<b>只需Redo 无需Undo</b>	<b>需要Redo 需要Undo</b>
Force	<b>无需Redo 无需Undo</b>	<b>无需Redo 只需Undo</b>

**日志/恢复策略**

### 7.3.三种日志类型

这一节老师讲的很快，很多地方存在疑惑（为什么要执行到哪个地方，检查点实际的作用）都不太清楚，只能盲目搬运 PPT 了。

- 判断确定每一个事务是否已完成：

- 已完成 (Redo 关注的) :

◦ < START T > … < COMMIT T >

- 未完成 (Undo 关注的) :

- < START T > … < ABORT T >
- < START T > …

- 检查点类型:

- 静止检查点:

- 周期性地对日志设置检查点。
- 停止接受新的事务，等到所有当前活跃事务提交或终止，并在日志中写入了 COMMIT 或 ABORT 记录。
- 最后将日志刷新到磁盘，写入日志记录 < CKPT >，并再次刷新日志。

- 非静止检查点:

- 在设置检查点时不必关闭系统，允许新事务进入。
- 写入一条 < START CKPT( $T_1, \dots, T_k$ ) >，其中  $T_1, \dots, T_k$  是所有活跃的未结束的事务。
- 继续正常的操作，直到  $T_1, \dots, T_k$  都完成时，写入 < END CKPT >。

- Undo 型日志:

- 对应 Steal + Force 策略

- 对于任一事务 T，按下列顺序向磁盘输出 T 的日志信息:

- 首先，< T, X, v > 被写到日志中，v 为 X 的旧值。
- 其次，OUTPUT(X)
- 最后，< COMMIT T > 或 < ABORT T > 被写到日志中

将事务改变的所有数据写到磁盘前不能提交该事务。

- 从日志的尾部开始按日志记录的反序，处理每一日志记录，撤销未完成事务的所有修改，处理到检查点为止。
- 如果是在 < COMMIT T > 后发生故障，无需对 T 做任何处理，因为 T 已经完成它要做的所有动作了，满足原子性。
- 反之，T 所有动作带来的影响都要撤销掉，通过 < T, X, v > 还原。
- 可能频繁地写磁盘，导致性能下降。

- Redo 型日志:

- 对应 No Steal + No Force 策略

- 对于任一事务 T，按下列顺序向磁盘输出 T 的日志信息:

- 首先，< T, X, v > 被写到日志中，v 为 X 的新值。
- 其次，< COMMIT T > 或 < ABORT T > 被写到日志中
- 最后，OUTPUT(X)

与 Undo 写入日志的顺序不一样。

- 从日志的起始位置（检查点）开始按日志记录的正序处理每一日志记录，重做已提交事务的所有修改。

在检查点处，要将之前所有已提交的事务写回磁盘（Undo 不用是因为它的 OUTPUT 在提交前做完）。

- 如果是在 < COMMIT T > 前发生故障，无需对 T 做任何处理，因为 T 并没有将影响写入数据库（OUTPUT 还没执行）。
- 反之，通过 < T, X, v > 更新数据库。
- 数据必须在 Commit 后才可见，灵活性差。

- Undo/Redo 型日志:

- 对应 Steal + No Force 策略

- 对于任一事务 T，按下列顺序向磁盘输出 T 的日志信息:

- 首先，< T, X, u, v > 被写到日志中，u 为 X 的旧值，v 为 X 的新值。
- (< COMMIT T > 或 < ABORT T >) 或 OUTPUT(X) 被写到日志中

- OUTPUT(X) 或 (< COMMIT T > 或 < ABORT T >) 被写到日志中

无所谓 OUTPUT 和 COMMIT 谁先写，很灵活。

- 先自后向前地撤销所有未提交的事务，再自前向后地重做所有已提交的事务。