数据库系统

> RyanFcr
>
> 笔记的基于前辈的笔记上加以修改的，在此鸣谢前辈

- 高云君老师，2021-2022春夏学期
  - IP: 10.214.6.33
    User: db50 – db59　use500 – user599
    Password of sa: sa
    Password of user: userXXX, e.g., user500 for user500
  - Project:30%
    - 数据库系统实验要求（3个）
    - 图书管理系统实验要求
    - MINI SQL系统设计与实现（说明团队做了什么，自己的part做了什么）
  - Assignments/Exercises——10%，作业命名：第几次_学号__名字
  - Course quizs——10%
  - Final Eaxm——50%
    - 12个选择题
    - 5个大题
  - 课件：10.214.131.93，user:csdbs，password:csdbs，port:21
- 复习建议：
  - 这门课的PPT非常乱，可以看看我整理的笔记，对PPT中的内容进行了适当的删改
  - 多刷刷历年卷，平时分不出问题，刷完题上个4还是不算难的
  - 平时作业好好做，尤其是miniSQL要早点完成，否则期末会非常爆炸

数据库系统

第一部分：基本概念和关系代数

1.0 课程绪论

1.0.1 数据库

1.0.2 View of Data

1.0.2.1 Level of Data Abstraction

1.0.2.2 Schemas and Instances

1.0.2.3 Physical Independence vs. Logical Independence

1.0.2.4 Data Model

1.0.3 Database Language

1.0.3.1 Data Definition Language (DDL)

1.0.3.2 Data Manipulation Language (DML)
1.0.3.3 SQL
1.0.3.4 Application Program Interface（API）
1.0.4 Database Design
1.0.5 Database Architecture
1.0.5.1 Storage Manager 存储管理
1.0.5.2 Query Processor 查找处理
1.0.6 Transaction Management
1.1 关系型数据库 Relational Database
1.2 基本概念和结构
1.3 Keys键
1.4 Relational algebra关系代数
1.4.1 基本操作
1.4.2 扩展运算
1.4.3 数据库的修改
1.5 实例
第二部分：SQL
2.1 SQL基本概念
2.2 DDL
2.3 SQL查询
2.4 SQL插入，删除，更新
2.5 SQL view 视图, index 索引
2.6 SQL Data Types and Schemas
2.7 Integrity 完整性控制
2.8 Authorization
2.9 Embedded SQL
2.10 ODBC and JDBC
2.11 SQL函数
第三部分：ER模型和Normal Form(范式)
3.1 E-R模型
3.1.1 ER设计问题
3.2 E-R Diagram
3.3 Normal Form 范式
3.3.1 数据库设计的目标
3.3.2 First Normal Form 第一范式
3.3.3 Functional dependency 函数依赖
3.3.4 闭包
3.3.5 BCNF/3NF
3.3.6 最小覆盖
第四部分: 数据库设计理论
4.1 存储和文件结构
4.1.1 磁盘 Magnetic Disks
4.1.2 File organization 文件组织
4.2 B+树索引
4.2.1 索引

4.2.2 B+树索引
4.2.3 文件索引
4.2.4 总结：存储结构和B+树的计算
4.3 查询处理 QueryProcess
4.3.1 select的cost估计
4.3.2 sort和join的cost估计
4.4 查询优化 Query Optimization
4.4.1 等价关系代数表达式
4.4.2 关系表达式的转换
4.4.3 cost的估计
第五部分：事务处理
5.1 事务和并发控制
5.1.1 基本的概念
5.1.2 事务的并发执行
5.1.3 Concurrency Control 并发控制
5.2 Recovery System 事务恢复
5.2.1 log-based Recovery基于日志的恢复
5.2.2 ARIES Recovery Algorithm——Aries恢复算法

第一部分：基本概念和关系代数

1.0 课程绪论

1.0.1 数据库

> 数据库：长期存储在计算机内、有组织的、可共享的数据集合
>
> Database Management System (**DBMS**)：(Database) + A set of programs used to access, update and manage the data in database.
>
> 把**数据和程序**分开，方便不同的程序操作数据
>
> - Efficiency and scalability in data access.
> - Reduced application development time.
> - Data independence (including physical data independence and logical data independence).
> - Data **integrity** and **security**.
>   - 数据完整性：数据有非显式的特征，方便加或者更改特征
> - Concurrent access and robustness (i.e., recovery). 支持大规模并非
> - Atomicity 原子性
> - Data Persistence 数据持久性

FPS：File-processing system is supported by a conventional Operating System (OS).

七个问题

- Data redundancy and inconsistency
  - 数据命名不统一，减少数据冗余。只有一个地方有数据
- Difficulty in accessing data
- Data isolation — multiple files and multiple formats
- Integrity problems
- No atomicity of updates
  - 不是原子性，数据库的操作是原子性的
- Difficult to concurrent access by multiple users
- Security problems (i.e., Right person uses right data)

> 这些数据库可以解决

1.0.2 View of Data

1.0.2.1 Level of Data Abstraction

> 数据有三个抽象层级，一层改变，不太影响其他两层，具有强适应能力

Different usage needs different level of abstraction.

- Physical level: describes how a record is stored.
- Logical level: describes data stored in database, and the relationships among the data on upper level.
- View level: application programs hide details of data types. Note that views can also hide information (e.g., employee's salary) for security purposes.

1.0.2.2 Schemas and Instances

> Schema（模式） – the structure of the database on different level
>
> - physical schema
> - logical schema
> - subschema
>
> Instance（事例） – the actual content of the database at a particular point in time
>
> > Similar to types and variables in programming languages
> > **type ↔ schema, variable ↔ instance**

1.0.2.3 Physical Independence vs. Logical Independence

> Ability to modify a schema definition at one level without affecting a schema definition at a higher level.
>
> 不会互相影响，独立性，适应变化
>
> - Physical data independence – the ability to modify the physical schema without changing the logical schema.
> - Logical data independence – protect application programs from changes in logical structure of data.

1.0.2.4 Data Model

> Different level of data abstraction needs different data model to describe
> 数据模型
>
> - Entity-Relationship model
> - Relational model
> - Other models:
>   - Object-oriented model
>   - Semi-structured data models (XML)
>   - Older models such as network model, hierarchical model, etc.

1.0.3 Database Language

1.0.3.1 Data Definition Language (DDL)

> Specification notation for **defining** the database schema.
>
> metadata元数据：记录数据的数据

1.0.3.2 Data Manipulation Language (DML)

> Language for accessing and **manipulating** the data organized by appropriate data model
>
> AKA query language
>
> - Procedural 过程式：用户可以指明一列可顺序执行的运算，以表示相应的计算过程
>   - 顺序结构
>   - 循环结构
>   - 分支结构

> - Declarative（nonprocedural）陈述式：非过程语言编写的程序可以不必遵循计算机执行的实际步骤，使人们无须关心问题的解法和计算过程的描述。在非过程语言中，只要指明输入记录、所要完成的加工以及输出形式，便能得到所要求的输出结果，其余工作全部由系统来完成。

1.0.3.3 SQL

SQL = DDL + DML + DCL(Data Control Language)

> SQL is the most widely used non-procedural query language.
>
> Structured Query Language (SQL) --- IBM. System R, Called Structured English QUEry Language (SEQUEL), 1975.

1.0.3.4 Application Program Interface（API）

> ODBC，JDBC

除了嵌入式的SQL

我们可以用API提供一个把SQL接入的接口

1.0.4 Database Design

> 建模的模型 区别于数据的模型
>
> E-R model
>
> - Entities (objects) 把实体集合建立起来
>   - E.g., customers, accounts, bank branch.
>   Entities are described by a set of attributes.
> - Relationships between entities 建立实体之间的关系
>   - E.g., Account A-101 is held by customer Johnson.
>   Relationship set depositor associates customers with accounts
>
> Normalization Theory 规范化理论

1.0.5 Database Architecture

1.0.5.1 Storage Manager 存储管理

> 在硬盘上的实现
>
> Storage Manager is a program module that provides the **interface** between the **low-level data** stored in the database and the application programs and queries submitted to the system.
>
> 缓冲管理：提供一块buffer比调用硬盘更快，需要对其进行管理

> B+树，多路查找，索引

1.0.5.2 Query Processor 查找处理

> Query Processor includes DDL interpreter, DML compiler, and query processing.
>
> 查询优化
>
> - Parsing and translation
> - Optimization
> - Evaluation

1.0.6 Transaction Management

> 事务并发

1.1 关系型数据库 Relational Database

关系型数据的一些基本特点

- 关系型数据库是一系列**表的集合**
- 一张表Relation是一个基本单位
- 表中的一行表示一条关系Relationship（元祖）
- 表中的一列表示一个属性

1.2 基本概念和结构

- a relation r is a subset of $D_1 \times D_2 \times \ldots \times D_n$，一条relation就是其中的一个n元的**元组(tuple)**
  - 一个D_i的笛卡尔积的子集

> 关系是n个元祖的**集合**

- attribute属性，指表中的**列名**
  - attribution type 属性的类型，属性是基本类型，不能是数组
  - attribute value 属性值，某个属性在某条relation中的值
    - 关系型数据库中的属性值必须要是**atomic**的，即不可分割的
    - domain：属性值的值域，**null是所有属性的domain中都有的元素**，代表不存在，但是null值会造成一些问题/复杂
- Relation Schema 关系模式

  > 描述了关系的结构
  >
  > Variable <——> relation

> Variable type <——> relation schema
>
> Variable value <——> relation instance

- $R = (A_1, A_2, \ldots, A_n)$ 就是Relation Schema，其中A_i是一系列属性，关系模式是对关系的一种**抽象**
- r(R)表示关系模式R中的一种关系，table表示这个关系当前的值(**关系实例**)

  > Relation instance：关系实例

  - 每个关系r中的元素是table中的一行
  - 不过经常用相同的名字命名关系模式和关系
- 关系是**无序**的，列的顺序是需要考虑的，关系中行和列的顺序是irrelevant的

1.3 Keys键

- super key超键：能够**唯一标识**元组的属性集，即对于每一条关系而言超键的值是唯一的
  - 超键可以是多个属性的组合
  - 如果A是关系R的一个超键，那么(A, B)也是关系R的一个超键
  - 超键的"唯一标识"各个元组是**可以有冗余信息**的
- candidate key候选键：**不含多余属性**的超键
  - 如果K是R的一个超键，而**任何K的真子集不是R的一个超键**，那么K就是R的一个候选键
  - 最小的超键
- primary key主键：
  - 这么多候选键里，数据库管理员**指定**的元组标识的一个候选键，**不能是null值**
- foreign key外键：用来描述两个表之间的关系，可以有空值（像C语言里的指针）
  - 如果关系模式R1中的一个属性是另一个关系模式R2中的一个**主键**，那么这个属性就是R1的一个外键
    - a foreign key from r1 referencing r2
      - r1的外码值一定要在r2中实际存在
    - 指向r2
    - 参照要有一致性

1.4 Relational algebra关系代数

> 关系代数(关系的集合，进行操作仍然是关系，闭空间)
>
> 可以和sql互相转化

1.4.1 基本操作

> 六个基本操作：

- Select 选择：$\sigma_p(r) = \{t|t \in r \wedge p(t)\}$
  - 筛选出所有满足条件p(t)的元素t，选择的是行
  - 会由一些op组成
- Project投影（纵向的选择）：$\prod_{A_1,A_2,\cdots,A_k}(r)$
  - 运算的结果是原来的关系r中各列只保留属性$A_1, A_2, \ldots, A_k$ 后的关系
  - 会**自动去掉重复**的元素，因为可能投影的时候舍弃的属性是可以标识关系唯一性的属性
- Union 并操作：$r \cup s = \{t|t \in r \vee t \in s\}$
  - 两个关系的属性个数必须**相同**，属性的domain是**可兼容的**
  - 各属性的domain必须是可以比较大小的compatible
  - 会**自动去掉重复的**

> 交不是基本操作，交操作可以通过并和差表示

- Set difference 差操作：$r - s = \{t|t \in r \wedge t \notin s\}$
  - 把两者都有的去掉
  - 两个关系的属性个数必须**相同**，属性的domain是可兼容的
  - 各属性的domain必须是可以比较大小的
- Cartesian-Product笛卡尔积：$r \times s = \{tq|t \in r \wedge q \in s\}$

  > 当涉及到多张表的时候，再笛卡尔积
  >
  > 自身比较的时候用笛卡尔积。求最大，找小于，再所有的减一下

  - 两个关系必须是**不相交**的，如果相交则需要对结果中重复的属性名进行**重命名**
  - 笛卡儿积运算的结果关系中元组的个数应该是rs的个数之**乘积**
- Renaming重命名：$\rho_{X(A_1,A_2,\ldots,A_n)}(E)$
  - 将E重命名为x, 让一个关系拥有多个别名，同时X可以写为 $X(A_1, A_2, \ldots, A_n)$ 表示对**属性也进行重命名**
  - 类似于C++中的引用

1.4.2 扩展运算

- 扩展运算: 可以用前面的六种基本运算得到，不增强关系代数的表达能力
  - Intersection 交运算$r \cap s = \{t | t \in r \wedge t \in s\} = r - (r - s)$
    - 两个关系的属性个数必须**相同**，属性的domain是可兼容的
    - 各属性的domain必须是可以比较大小的compatible
    - $r \cap s = r - (r - s)$
  - Natural-Join 自然连接：$r \bowtie s = \prod(\sigma(r \times s))$
    - 两个关系中**同名**属性在自然连接的时候当作**同一个属性**来处理（选出同名属性值相同的元祖+笛卡尔积）
      - 先笛卡尔积，再选择，再投影（只是去重的作用）
      - Natural Join是associate and commutative，满足结合律和交换律
    - **Theta join** 满足某种条件的合并：$r \bowtie_\theta s = \sigma_\theta(r \times s)$
      - 自然连接 + 条件
  - Outer-Join外部连接，分为左外连接，右外连接，全外连接
    - 用于应对一些**信息缺失**的情况(有null值)
    - 左外连接⟕
      - 左边的表取全部值按照关系和右边连接，右边不存在时为空值
    - 右外连接⟖
      - 右边的表取全部值按照关系和右边连接，不存在为**空值**
    - Full join左右全上，不存在对应的就写成空值
  - Division除法：$r \div s = \{t | t \in \prod_{R-S}(r) \wedge \forall u \in s(tu \in r)\}$
    - 如果$R = (A_1, A_2, \ldots, A_m, B_1, \ldots, B_n) \wedge S = (B_1, \ldots, B_n)$ 则有$R - S = (A_1, A_2, \ldots, A_m)$
      - 把r，s中同名属性，**相同值**情况下的元祖拿出来，再project，去掉同名属性
      - 先把$R - S$中的情况分类，再看看是否满足S的要求，满足的话就保留，再project
    - 当涉及到求**全部**之类的查询，常用除法
    - $q = r \div s$，q是最大的满足$q \times s \subseteq r$的关系
    - $r \div s = \prod_{R-S}(r) - \prod_{R-S}((\prod_{R-S}(r) \times s) - \prod_{R-S}(r))$
  - Assignment声明操作，类似于变量命名用$\leftarrow$ 可以把一个关系代数操作进行命名
    - 简化

> 优先级：

> 1. Project
> 2. Select
> 3. Cartesian Product (times)
> 4. Join, division
> 5. Intersection
> 6. Union, difference

- Generalized Projection扩展的投影$\prod_{F_1,F_2,\cdots,F_k}(r)$，F是算术表达式
- Aggregation operations聚合操作
  - 基本形式：$_{G_1,G_2,\ldots,G_n}\mathcal{G}_{F_1(A_1),\ldots,F_n(A_n)}(E)$
  - G是聚合（可以理解成分类）的标准，对于关系中所有G值相同的元素进行**聚合**，F( )是聚合的运算函数，A（）是属性
    - 聚合的意思指相同的分到一组进行F运算操作
  - 常见的有**SUM/MAX/MIN/AVG/COUNT**（非空值的个数）
  - 也可以不分组，不写G
  - 支持直接改名$_{G_1,G_2,\ldots,G_n}\mathcal{G}_{avg(salary)as\quad avg_sal}(E)$
    - 不改名的话是没有名字的

> Null值进行运算，最后都是Null

1.4.3 数据库的修改

- Modification of the Database数据库的更改

> 所有操作都由赋值来表示
>
> 先后顺序需要考虑

  - Deletion
    - $r \leftarrow r - E$
  - Insertion
    - $r \leftarrow r \cup E$
  - Updating
    - $r \leftarrow \prod_{F_1,F_2,\ldots,F_k}(r)$
    - F是一些更新的函数

> 一般我们不需要去掉重复的——>Multiset多重集
>
> - 去掉重复的浪费算力
>   - 一般代价是O（logn）
> - 有时候我们需要重复的数据来反应真实

- Multiset
  - selection：不去重
  - projection：不去重
  - cross product
  - set operators
    - union：m+n copies
    - intersection：min（m，n） copies
    - difference：max（0，m-n）copies

1.5 实例

❑ Example 5: Find the names of all customers who have a loan at the Perryridge branch.

Query 1: $\Pi_{customer\text{-}name}(\sigma_{branch\text{-}name='Perryridge'}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(borrower \times loan)))$

Query 2: $\Pi_{customer\text{-}name}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(borrower \times (\sigma_{branch\text{-}name='Perryridge'}(loan))))$

Query 2 is better.

loan(*loan-number*, *branch-name*, *amount*)
borrower(*customer-name*, *loan-number*)

- Example 6: Find the names of all customers who have loans at the Perryridge branch but do not have an account at any branch of the bank.

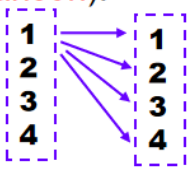
Query 1: $\Pi_{customer\text{-}name}(\sigma_{branch\text{-}name='Perryridge'}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(borrower \times loan))) - \Pi_{customer\text{-}name}(depositor)$

Query 2: $\Pi_{customer\text{-}name}(\sigma_{borrower.loan\text{-}number = loan.loan\text{-}number}(borrower \times (\sigma_{branch\text{-}name='Perryridge'}(loan)))) - \Pi_{customer\text{-}name}(depositor)$

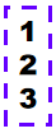
Query 2 is better.

loan(*loan-number*, *branch-name*, *amount*)
borrower(*customer-name*, *loan-number*)
depositor(*customer-name*, *account-number*)

- Example 7: Find the largest account balance (i.e., self-comparison).
  - Step 1: Rename *account* relation as *d*.
  - Step 2: Find the relation including all balances except the largest one.



$\Pi_{account.balance}(\sigma_{account.balance<d.balance}(account \times \rho_d(account)))$



d

  - Step 3: Find the largest account balance.

$\Pi_{balance}(account) - \Pi_{account.balance}(\sigma_{account.balance<d.balance}(account \times \rho_d(account)))$

account(*account-number*, *branch-name*, *balance*)

- Example 1: Find all customers who have an account from at least the "Downtown" and the "Uptown" branches.

Query 1: $\Pi_{customer\text{-}name}(\sigma_{branch\text{-}name='Downtown'}(depositor \bowtie account)) \cap \Pi_{customer\text{-}name}(\sigma_{branch\text{-}name='Uptown'}(depositor \bowtie account))$

Query 2: $\Pi_{customer\text{-}name,\ branch\text{-}name}(depositor \bowtie account) \div \rho_{temp(branch\text{-}name)}(\{('Downtown'), ('Uptown')\})$

depositor(*customer-name*, *account-number*)
account(*account-number*, *branch-name*, *balance*)

- Example 2: Find all customers who have an account at all branches located in Brooklyn city.

$$\Pi_{customer\text{-}name,\ branch\text{-}name}(depositor \bowtie account) \div \Pi_{branch\text{-}name}(\sigma_{branch\text{-}city='Brooklyn'}(branch))$$

branch(*branch-name*, *branch-city*, *assets*)
depositor(*customer-name*, *account-number*)
account(*account-number*, *branch-name*, *balance*)

- Example 3: 查询选修了全部课程的学生学号和姓名。
  - 涉及表: 课程信息*course*(*cno*, *cname*, *pre-cno*, *credits*), 选课信息 *enrolled*(*sno*, *cno*, *grade*), 学生信息*student*(*sno*, *sname*, *sex*, *age*)
  - 当涉及到求“全部”之类的查询，常用“除法”。
  - 找出全部课程号：$\Pi_{Cno}(Course)$
  - 找出选修了全部课程的学生的学号：$\Pi_{Sno,\ Cno}(enrolled) \div \Pi_{cno}(Course)$
  - 与*student*表自然连接（连接条件*Sno*）获得学号、姓名：$(\Pi_{Sno,\ Cno}(enrolled) \div \Pi_{cno}(Course)) \bowtie \Pi_{Sno,Sname}(student)$

第二部分：SQL

2.1 SQL基本概念

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

> Structured English Query Language (SQL)
>
> SQL = DDL + DML + DCL(Data Control Language)

2.2 DDL

- DDL：Data-definition language
  - 定义数据结构：schema for each relation
  - 定义值域：domain
  - 完整约束Integrity constraints
  - 物理层面的结构、信息
- SQL支持的数据类型
  - char（n）：定长字符串
  - varchar（n）：可变长度字符串
  - int
  - smallint
  - numeric(p,d)：和浮点数相对，定长，用户精度p位，小数点后d位
  - real、double precision：单精度双精度

    - float（n）：至少n位
    - null-value：空也是一种类型
- Build-in
    - date：年月日
    - time：时分秒
    - timestamp：年月日时分秒
    - interval：时间间隔类型
        - interval可以和date/time/timestamp values相加
        - data,time funtions:
            - current_date(),current_time()
            - year(x),month(x),day(x),hour(x),minute(x),second(x)
    - 所有的数据类型都支持null作为属性值，可以在定义的时候声明一个属性的值not null
- 创建数据库`create database <name>;`，再`use <name>;`
- 创建数据表`create table;`
    - 创建表的语法

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

    - `integrity-contraint`完整性约束
        - 可以**指定primary key** $(A_1 \dots A_n)$
        - **foreign key** $(A_m, \dots, A_n)$**references xxx**
            - 如果引用被删了
                - 一起删掉
                - 置空
                - 强制不允许删除，先处理好再删除
        - **not null**
        - Check (P), where P is a predicate
- 删除数据表`drop table R`
    - 元数据都删除了
- 更新数据表的栏目`alter table`

- `alter table R add A D` 添加一条新属性
  - 其中A是属性名，D是A的domain
- `alter table R drop A` 删除A属性
  - 许多数据库不支持删除操作

2.3 SQL查询

> DML：data-manipulation language；提供查询、插入、删除、更新
>
> SQL很重要

- SQL查询的基本形式：select语句

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

  - 上述查询等价于$\prod_{A_1,A_2,\cdots,A_k}(\sigma_p(r_1\times r_2\times\cdots\times r_m))$
  - SQL查询的结果是一个关系（一张表）
- select子句的一些**细节**
  - `select * from xxx` 表示获取**所有属性**，事实上*是正则表达式，表示可能有的所有内容，从后面的内容来看，select语句确实是支持正则表达式
  - SQL中的**保留字**对于**大小写不敏感**，不能使用`-`
  - 去除重复：`select distinct`，防止重复丢失的办法`select all`
  - select子句中的表达式支持基本的**四则运算**(加减乘除)，比如

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

  - where子句中：
    - 支持`and or not`等逻辑运算
    - 支持`between and`来查询范围
    - 支持元祖比较
      - where (instructor.ID, dept_name) = (teaches.ID, 'Biology')
      - 或者直接使用`natural join`，但要避免名字一样，但是含义不一样的情况
  - from 子句：
    - from 可以选择多个表，此时会先将这些表进行笛卡儿积的运算，再进行select

- 元组变量：可以从多个表中select满足一定条件的几个不同属性值的元组

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

- 重命名操作，可以通过`old_name as new_name`进行重命名
  - for simplification
  - for discrimination
- 字符串：支持**正则表达式**匹配，用`like regrex`的方式可以进行属性值的正则表达式匹配。这个是区分大小写的。
  - 正则表达式的用法
    - `%`：代表一串字符串
    - `_`：代表一个字符
    - 转义再 + `\`
      - like ‘100 #%’ **escape** ‘#’：转义字符可以自己调整
  - 还支持连接（using ||）
  - 从大写转移到小写
  - 找字符串长度，抽取最小子串

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


'___'刚好三个
'___%'至少三个
```

- 将输出的结果**排序**
  - `order by 属性名 asc/desc`
    - desc降序
    - asc升序
  - 可以按照多个排序，先按第一个排，第一个相同再按第二个排
- limit：限制选择返回的行数

    - `limit offset , row_count`
- 多重集：和上面一样
- 集合操作：
    - 可以用`union/intersect/except`等集合运算来**连接两条不同的查询**（是去重的）
    - 与，交，差
    - 如果想不去重的话，可以在后面加`all`
- 聚合操作：
    - 支持的操作有avg/min/max/sum/count，获取的是表中的统计量

```
select dept_name,avg(salary) as avg_salary
from instructor
group by dept_name;//对每一组进行输出，一定要是select里面的,且s变量elect里面不能有其他的不在聚合函数里的
```

    - `count(*)`返回表中的非空记录数，聚合函数忽略空值
    - 可以加`distinct`
    - 事实上SQL语句的聚合操作和关系代数中的聚合运算是完全对应的，关系代数中的聚合运算表达式$_{G_1,G_2,\ldots,G_n}\mathcal{G}_{F_1(A_1),\ldots,F_n(A_n)}(E)$对应的SQL语句是

```
select G1,G2,…,Gn,F1(A1),…,Fn(An)
from E
group by G1,G2,…,Gn;
```

    - 聚合操作的SQL语句书写可以在末尾用`having xxx`来表示一些需要聚合操作来获得的条件（**聚合函数不能直接用于where语句**），对组进行筛选，比如

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

    - 如果组只有null值，count返回0，其他聚合操作返回null
- Null values 空值
    - 属性值可以为null，当然也可以在定义数据表的时候规定哪些元素不能为空
    - null的算术表达式 = null
    - null的比较 = null

    - OR: (unknown or true) = true
      (unknown or false) = unknown　　(unknown or unknown) = unknown
      AND: (true and unknown) = unknown
      (false and unknown) = false
      (unknown and unknown) = unknown
      NOT: (not unknown) = unknown
    - 检查是不是null，要用`is null`，而不是`=`
- Summary
  - The format of SELECT statement:
    SELECT <[DISTINCT] c1, c2, ...>
    FROM <r1, ...>
    [WHERE]
    [GROUP BY <c1, c2, ...> [HAVING]]
    [ORDER BY <c1 [DESC] [, c2 [DESC|ASC], ...]>]

> From → where → group (aggregate) → having → select → distinct → order by

- Nested Subquery 嵌套查询
  - 对于查询

```
select A1,A2,…,An
from r1,r2,…,rn
where P
group by G
having W
```

    其中的A，r，P，W都可以被替换嵌入为一个**子查询**
  - 集合关系：用`in/not in +子查询`来判断否些属性是否属于特定的集合中
    - `some+子查询`用于判断集合中是否存在满足条件的元组，用来判断存在性
    - `all+子查询`可以用来筛选**最值**
    - `exists+子查询`判断子查询的结果是否不为空
    - `not exists+子查询`判断是否为空集，空就返回1
      - 可以替代"除法"的情况，全部
    - `unique+子查询`判断是否是唯一的
      - 空集也是唯一的，如果想保证必须存在，可以再加一个`exist`
      - 是个集合，不是多重集

        - `not unique`
    - *with子句：对子查询定义一个变量名，可以在之后调用，引出中间变量，一个临时的表
    - scalar子查询：用于需要一个值作为查询结果的时候
    - join子句：可以对若干张表进行各种join之后再查询
        - join condition：定义哪些tuples是match的
        - 经常在`from` clause里用
        - `natural join` 自然连接
        - `A join B on(xxx)`
        - Join types
            - inner join/native join
            - left outer join
            - right outer join
            - full outer join
        - ```
          natural                 //自然连接
          on<predicate>           //条件连接
          using (A1,A2,……)        //等值连接
          ```

- 自然连接：R natural {inner join, left join, right join, full join} S
- 非自然连接：R {inner join, left join, right join, full join} S
    - on <连接条件判别式>
    - using (<同名的等值连接属性名>)

> 非自然连接，容许不同名属性的比较，且结果关系中不消去重名属性。
>
> 使用using的连接类似于natural连接，但仅以using列出的公共属性为连接条件

2.4 SQL插入，删除，更新

- 插入：`insert into <table|view> [(c1,c2,c3……)] values(e1,e2,e3 );`

  `insert into <table|view> (c1,c2,c3……) select (e1,e2,e3 ) from;`
    - `[]` 的意思是可以不要
    - 插入要么**记住顺序**，要么在`table_name`后写属性，然后对应的在后面加
    - 有空值就null
    - 可以用**select查询子句**得到的结果作为values，此时可以同时插入多条结果
        - `insert into table1 select * from table2`
        - 只会插一次
          ```

- 删除：`delete from <table|view> where <condition`
  - 删除条件中包括avg的情况下，需要先固定取出avg，不然avg会改变
  - delete后面只能跟一个关系
- 更新：`update <table|view> set xxx where xxxxx`
  - case 子句：用于分类讨论
  - 注意case的条件顺序

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

2.5 SQL view 视图, index 索引

- 视图：一种**只显示数据表中部分属性值**的机制
  - 不会在数据库中重新定义一张新的表，而是**隐藏**了一些数据，虚表
  - 安全隐私
  - 权限
  - 抽象，Independence，适应变化能力
  - 创建视图的定义语句：

    > 建立在单个基本表上的视图，且视图的列对应表的列；View是虚表，对其更新的操作都将转化成对基表的操作，只有行列视图，可更新数据

    - xxx是视图的名称，内容是从某个table中select出的

```
create view xxx as
    select c1，c2……from……(a subquery)
```

```
drop view <v_name>
```

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

    - 语法`create index index_name on table_name(attribute)`
    - 在对应的表和属性中建立索引，加快查询的速度
    - 索引
        - CREATE INDEX ON ();
        - CREATE UNIQUE INDEX ON ();
            - 独一无二的
        - To drop an index：
          DROP INDEX ON
        - 也可以用alter
- Transactions 事务
    - 一系列查询等操作的集合（**原子性的**）
    - A transaction is a sequence of queries and data update statements executed **as a single logical unit**
    - ```
      SET AUTOCOMMIT = 0;//设定取消自动执行
      ```
    - ```
      COMMIT;
      UPDATE account SET balance=balance-200WHERE ano ='1003'
      UPDATE account SET balance=balance+200 WHERE ano='1004'
      COMMIT;
      ```
    - Transactions are started implicitly and terminated by one of
        - COMMIT WORK: makes all updates of the transaction permanent in the database.
        - ROLLBACK WORK: undoes all updates performed by‘ the transaction.
    - Atomic transaction 原子事务：只能被**完全执行**或者**回滚**(roll back)

> ACID
>
> - Atomicity
> - Cosistency
> - Isolation
> - Durability

2.6 SQL Data Types and Schemas

- Type创建新类型
  - 创建自己的数据类型，但是**无约束条件**；`create type person_name as varchar(20)`
  - 删除就`Drop type person_name`
  - 有点像typedef
- Domain创建新类型
  - `create domain new_name + data type`(比如 char(20))
  - ```
    Create domain Dollars as numeric(12, 2) not null;
             Create domain Pounds as numeric(12,2);
    ```
  - domain可以设置约束条件，比如下面这一段domain定义表示degree_level只能在这三个中进行选择

    ```
    create domain degree_level varvhar(10)
    constraint degree_level_test
    check(value in ('Bachelors', 'Masters', 'Doctorate'));
    ```
  - Large-Object Types 大对象类型，分为blob(二进制大对象)和clob(文本大对象)两种，当查询需要返回大对象类型的时候，取而代之的是一个代表大对象的指针
    - blob存图片、视频
    - clob存文本
    - 更多情况是存目录，存指针（不然会很大）

    ```
    E.g., Create table students
                                        (sid char(10)
    primary key,
                                         name varchar(10),
                  gender char(1),
                                         photo blob(20MB),
                      cv clob(10KB))
    ```

2.7 Integrity 完整性控制

> 帮我们guarantee

- 单个关系上的约束
  - 主键 primary key（不可为空，不可以重复）， **unique**， not null，foreign key
    ```

- check子句：写在数据表的定义中
  - check(P) 检查某个属性**是否为特定的一些值**
  - 在更新或者插入的时候会检查

```
Create table branch2
                        (branch_name varchar(30) primary key,
                         branch_city varchar(30),
                                     assets integer not null,
                         check (assets >= 100))
```

- Domain constraints 值域的约束
  - 在domain的定义中加入check
  - 语法 `create domain domain_name constraints check_name check(P)`

```
Create domain hourly-wage numeric(5, 2) Constraint value-test check(value > = 4.00)
```

- **Referential Integrity** 引用完整性
  - 被引用表中主键和外键的关系
  - 参照关系中外码的值必须在被参照关系中**实际存在，或为null**.
    - 当已经存在关系的时候，不可删除
    - 不可修改
  - 在定义表的时候定义主键和外键进行约束

```
Create table depositor
    (customer-name  char(20),
     account-number char(10),
     primary key (customer-name, account-number),
     foreign key (account-number) references account,
     foreign key (customer-name) references customer);

```

- Cascading action级联：要一起更新，并发（是自己定义的）（每一层都需要定义，不然一旦发现没有定义，就会取消操作，回滚）
  - on update
  - on delete

```
Create table account (
            . . .
    foreign key (branch-name) references branch
        [ on delete cascade]
        [ on update cascade ]
        . . . );
```

- 对于整个数据库全局性的约束
  - Assertions
    - 对于数据库中需要满足的关系的一种**预先判断**
    - 每次更新都检查
    - `create assertion <assertion-name> check <predicate>` 下面是一段例子

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

```
E.g., if we require “the sum of all loan amounts for
each branch must be less than the sum of all account
balances at the branch”.

But SQL does not provide a construct for asserting:
        for all X, P(X)
So it is achieved in a round-about fashion, using:
        not exists X, such that not P(X)


Example 1: The sum of all loan amounts for each branch
must be less than the sum of all account balances at
the branch.

         CREATE ASSERTION sum-constraint CHECK
                (not exists (select * from branch B
                        where (select sum(amount)
from loan
```

```
                                        where
loan.branch-name = B.branch-name)
                                            > (select
sum(balance) from account
                                    where
account.branch-name =
                                B.branch-name)))

```

- Trigger触发器
  - 满足某些条件后系统自动执行的语句
  - ECA:Event,Condition,Action
  - 检测条件和动作
  - 时间节点的选择
    - **referencing old row/table as** 对旧的行进行操作，用于**删除和更新**
    - **referencing new row/table as** 对新的行进行操作，用于**插入和更新**
- trigger event触发事件
  - insert/delete/update等操作都可以触发设置好的trigger
  - 触发的时间点可以是before和after，触发器的语法如下

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

```
CREATE TRIGGER overdraft-trigger after update on account
    referencing new row as nrow for each row
    when nrow.balance < 0
        begin atomic
        insert into borrower
            (select customer-name, account-number from
depositor
             where nrow.account-number = depositor.account-
number)
            insert into loan values
            (nrow.account-number, nrow.branch-name, -
nrow.balance)
            update account set balance = 0
            where account.account-number = nrow.account-number
        end
```

- 删除就drop
- Triggers cannot be used to **directly implement external-world actions**, BUT Triggers can be used to **record actions-to-be-taken** in a separate table
  Have an external process that repeatedly scans the table, carries out external-world actions and deletes action from table.
- When Not To Use Triggers
  - 不要常用
  - Triggers were used earlier years for tasks such as:
    - Maintaining summary data (e.g., total salary of each department)
    - Replicating databases by recording changes to special relations (called change or delta relations) and having a separate process that applies the changes over to a replica.
  - There are better ways of doing these now:
    - Databases today provide built in materialized view facilities to maintain summary data;
    - Databases provide built-in support for replication.

2.8 Authorization

- 数据库中的四种权限 read,insert,update,delete
- view保护安全，也是一种权限
- Security specification in SQL 安全规范

- grant语句可以赋予用户权限 `grant <privilege list> on <relation name or view name> to <user list>`

```
grant select on instructir to U1,U2,U3
```

- `<user list>`可以是用户名，也可以是public(允许所有有效用户拥有这项权限)
- grant语句后面可以加`with grant option`，表示该用户拥有赋予其他用户这项权限的权力；**权限可以传递**
- revoke 权力回收
  - `revoke <privilege list> on <relation/view name> from <user list> [restrict|cascade]`从用户中回收权力
    - restrict：不收回之前传下去的，只收回你那个
    - cascade：收回之前传下去的东西
- role语句
  - `create role role_name`
  - 创造一种人群，然后可以赋予他们一些权限
  - 允许一类用户持有相同的权限
- reference 引用权限
- An **audit trail** is a log of all changes (inserts/deletes/updates) to the database along with information such as which user performed the change, and when the change was performed.
  - Used to track erroneous/fraudulent updates.
  - Can be implemented using triggers, but many database systems provide direct support.
  - ```
    AUDIT <st-opt> [BY <users>] [BY SESSION | ACCESS]
    [WHENEVER SUCCESSFUL | WHENEVER NOT SUCCESSFUL]
    ```
    - 当 BY 缺省，对所有用户审计。
    - BY SESSION每次会话期间，相同类型的需审计的SQL语句仅记录一次。
    - 常用的：table, view, role, index, ...
    - 取消审计：NOAUDIT ...(其余同audit语句)。
    ```

2.9 Embedded SQL

> 在其他语言中使用SQL
>
> - API
> - Embedded SQL

- SQL的功能不完备性。（计算，资源...）
- The SQL standard defines embeddings of SQL in a variety of programming languages such as Pascal, PL/I, Fortran, C, and Cobol.
- A language in which SQL queries are embedded is referred to as a **Host language** (宿主语言), and the SQL structures permitted in the host language comprise embedded SQL.
- SQLCA：通讯区（data struct）
- SQLDA：数据区（data struct）
- EXEC SQL statement is used to identify embedded SQL request to the preprocessor:

```
EXEC SQL <embedded SQL statement> END_EXEC

Note: This varies by language, e.g., the Java embedding
uses # SQL { …. }
```

```
main( )
{  EXEC SQL INCLUDE SQLCA; //声明段开始
    EXEC SQL BEGIN DECLARE SECTION;
      char account_no [11];    //host variables(宿主变量)声明
      int  balance;
   EXEC SQL END DECLARE SECTION;//声明段结束
   EXEC SQL CONNECT  TO  bank_db  USER Adam Using Eve;
   scanf (“%s  %d”, account_no, balance);
   EXEC SQL update account
                       set balance= balance+:balance
                       where account_number = :account_no;
   If (SQLCA sqlcode ! = 0)    printf ( “Error!\n”);
   else       printf (“Success!\n”);
}

```

```

 main( )
```

```
{  EXEC SQL INCLUDE SQLCA;
    EXEC SQL BEGIN DECLARE SECTION;
      char customer_name[21];
      char account_no [11];
      int  balance;
   EXEC SQL END DECLARE SECTION;
   EXEC SQL CONNECT  TO  bank_db  USER Adam Using Eve;

   EXEC SQL DECLARE account_cursor CURSOR for
              select account_number, balance
              from depositor natural join account
              where depositor.customer_name = :
customer_name;
            scanf (“%s”, customer_name);
          EXEC SQL open account_cursor;
          for (; ;)
          {   EXEC SQL fetch account_cursor into
:account_no, :balance;
               if (SQLCA.sqlcode!=0)
            break;
                printf( “%s  %d \ n”, account_no,
balance);
            }
            EXEC SQL close account_cursor;
}
```

- 多行操作——引入了csr游标
- 什么时候需要使用嵌入式SQL
  - 与用户交互时，出现特别复杂的检索结果难以用一条交互式SQL语句完成，此时需要结合高级语言中经常出现的顺序，分支和循环结构帮助处理
- Dynamic Sql

```
char *  sqlprog = “update account
   set balance = balance * 1.05                where
account_number = ?”EXEC SQL prepare dynprog  from
:sqlprog;char account [10] = “A-101”;EXEC SQL execute
dynprog using :account;

The dynamic SQL program contains a ?, which is a place
holder (占位符) for a value that is provided when the SQL
program is executed.
```

2.10 ODBC and JDBC

- Open DataBase Connectivity (ODBC, 开放数据库互连)
  - ODBC提供了一个公共的、与具体数据库无关的应用程序设计接口API 。它为开发者提供单一的编程接口，这样同一个应用程序就可以访问不同的数据库服务器。
- JDBC is a Java API for communicating with database systems supporting SQL.
  - Trail: JDBC Database Access (The Java™ Tutorials) (oracle.com)
- SQL注入攻击

2.11 SQL函数

```
Define a function that, given the name of a department,
returns the count of the number of instructors in that
department.
create function dept_count (dept_name varchar(20))
returns integer
begin
declare d_count integer;
select count (* ) into d_count
from instructor
where instructor.dept_name = dept_name
return d_count;
end

Find the department name and budget of all departments with
more that 12 instructors.

        select dept_name, budget
        from department
        where dept_count (dept_name ) > 1
```

```
SQL:2003 added functions that return a relation as a result
Example: Return all accounts owned by a given customer


    create function instructors_of (dept_name char(20) )
        returns table (     ID varchar(5),
                        name varchar(20),

                        dept_name varchar(20),
                        salary numeric(8,2))
```

```
    return table
    (select ID, name, dept_name, salary
     from instructor
     where instructor.dept_name = instructors_of.dept_name)


Usage
        select *
        from table (instructors_of ('Music'))
```

过程语句

类似于function

```
The dept_count function could instead be written as procedure:
    create procedure dept_count_proc (in dept_name
varchar(20),
         out d_count integer)
    begin
      select count(*) into d_count  from instructor  where
instructor.dept_name = dept_count_proc.dept_name
     end

Procedures can be invoked either from an SQL procedure or from
embedded SQL, using the call statement.

        declare d_count integer;
        call dept_count_proc( 'Physics', d_count);
    Procedures and functions can be invoked also from dynamic
SQL
```

```
While and repeat statements :
        declare n integer default 0;
        while n < 10 do
            set n = n + 1
        end while
              repeat
          set n = n  - 1
              until n = 0
        end repeat
```

```
For loop

declare n  integer default 0;
for r  as
    select budget from department
    where dept_name = ‘Music’
do
    set n = n - r.budget
end for
```

```
if boolean  expression
    then statement or compound statement
elseif boolean  expression
    then statement or compound statement
    else statement or compound statement
end if
```

```
create function registerStudent(
                in s_id varchar(5),
                in s_courseid varchar (8),
                in s_secid varchar (8),
                in s_semester varchar (6),
                in s_year numeric (4,0),
                out errorMsg varchar(100)
returns integer
begin
    declare currEnrol int;
    select count(*) into currEnrol
        from takes
        where course_id = s_courseid and sec_id = s_secid
            and semester = s_semester and year = s_year;
    declare limit int;
    select capacity into limit
        from classroom natural join section
        where course_id = s_courseid and sec_id = s_secid
            and semester = s_semester and year = s_year;
    if (currEnrol < limit)
        begin
            insert into takes values
                (s_id, s_courseid, s_secid, s_semester, s_year, null);
            return(0);
        end
    - - Otherwise, section capacity limit already reached
    set errorMsg = 'Enrollment limit reached for course ' || s_courseid
        || ' section ' || s_secid;
    return(-1);
end;
```

过程也可以用`c`写，直接call他

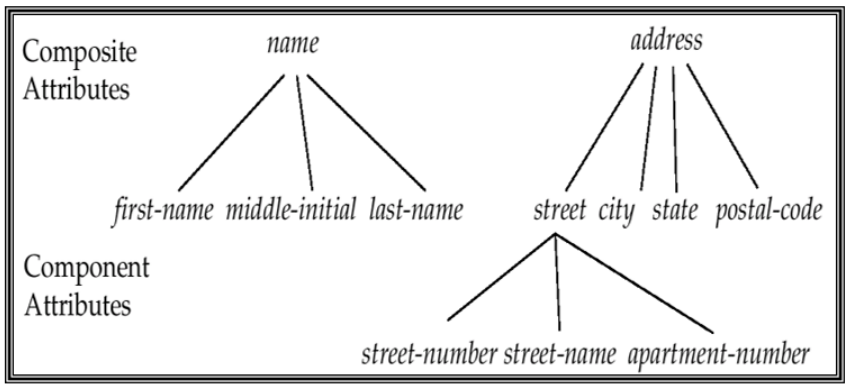
```
create function dept_count(dept_name varchar(20))
returns integer
language C
external name '/usr/avi/bin/dept_count'

```

第三部分：ER模型和Normal Form(范式)

3.1 E-R模型

- E-R模型由enitites(实体)和relation(关系)组成
- Entity set 实体集
  - 实体是一系列独特的对象，用一系列属性来表示
  - 同一类实体共享相同的Properties，实体集就是由同类型的实体组成的集合
  - 表示方法
    - **长方形**代表实体集合
    - 属性写在长方形中，**primary key用下划线**标注
  - 实体集中对于属性的定义和之前的几乎一样
    - Attribute types:
      - Simple and composite attributes (简单和复合属性，如sex, name).



        分量属性：子属性

      - Single-valued and multi-valued attributes(单值和多值属性).

        E.g., multivalued attribute:

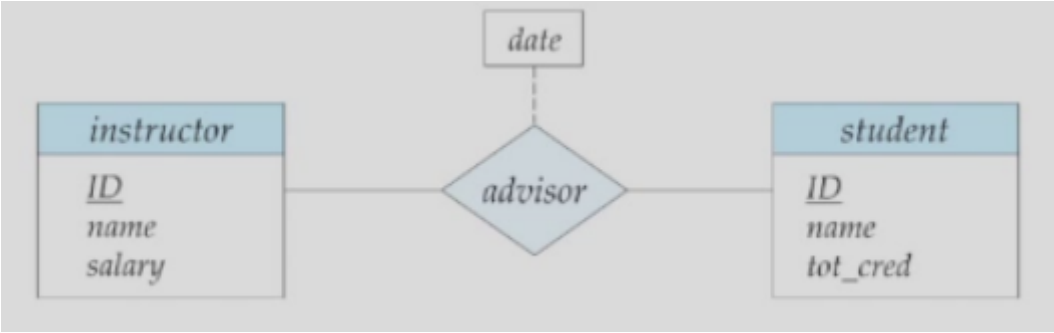
        用`{}`表示，比如{phone_number}

      - Derived attributes (派生属性).
        - Can be computed from other attributes, e.g., age, given date of birth.
        - versus base attributes or stored attributes (基属性，存储属性).
    - 实体集中属性定义可以存在组合与继承的关系（复合属性），下面是一个样例

      > Instructor
      >
      > ID
      >
      > name

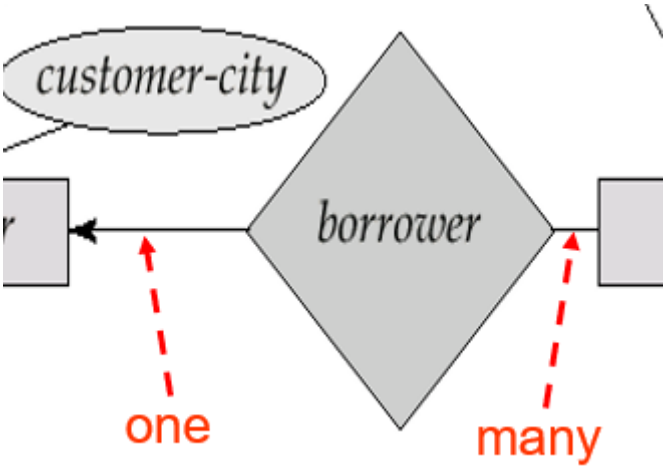
> first_name
> last_name
> address
> street
> street_number
> street_name
> apt_number
> city
> state

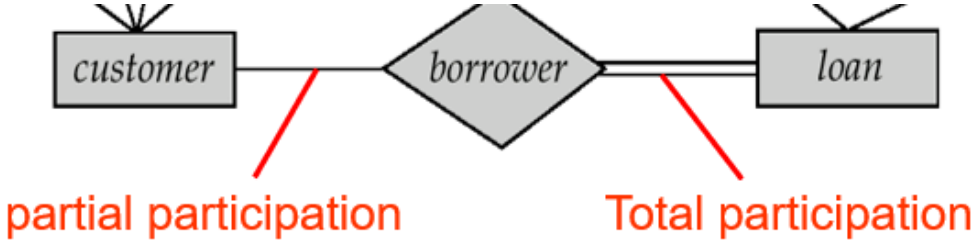
- Relationship set 关系集
  - 一个relationship是几个实体之间的联系，关系集就是**同类关系**之间构成的**集合**
  - 同样的实体集可以以不同的**角色**参与关系
  - **菱形**代表关系
  - 关系也可以有描述性属性，由一个虚线连出来



  - 一个relationship**至少需要两个及以上**的实体，一个关系集至少和两个实体集有关联
    - 一个关系集所关联的实体集的个数称为**degree**，其中以二元关系集为主
    - **我们常将多元关系转化成二元关系**
    - In general, any non-binary relationship can be represented using binary relationships by creating an artificial entity set.
- E-R model constraints 约束
  - mapping cardinalities **映射基数**
    - 二元关系中映射基数只有一对一，一对多，多对一，多对多（都是至多，可以零个）
    - E-R模型中表示映射关系：**箭头表示一，直线表示多**

> 对于一个borrower只有一个账户

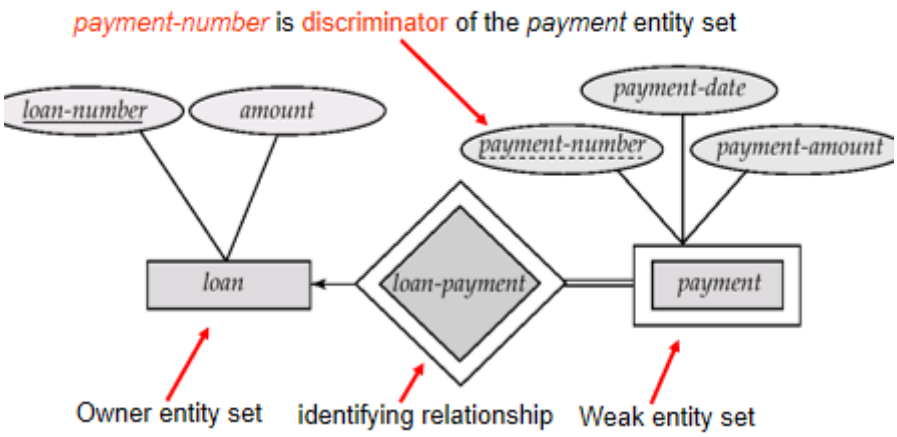
- 

  - 三元关系中：**箭头只能出现一次**，否则会出现**二义性** （多元会产生歧义）
- 参与度约束
  - total participation： 若一个实体集每一个实体至少参与一个关系（全部参与到关系中），要用**两条线**
  - partical participation：部分参与
  - 

- key约束：
  - super key
  - candidate key
  - primary key
- 弱实体集weak entity set：一些实体集的属性不足以形成主键，就是弱实体集，与之相对的是强实体集
  - 用于表示一些关系中的**依赖性**，弱实体集需要和**强实体集**关联才有意义
    - The existence of a weak entity set depends on the existence of a identifying entity set or owner entity set (标识实体集或属主实体集)
  - 弱实体集由强实体集的primary key和弱实体集的分辩符表示
  - 弱实体集的分辩符以**虚下划线**标明，关系弱实体集和标示性强实体集的联系集以**双菱形**表示

- the primary key of the strong entity set is not explicitly stored with the weak entity set, since it is implicit in the identifying relationship.
  不需要**存**强实体集的key

  Primary key for *payment*(*loan-number*, *payment-number*)

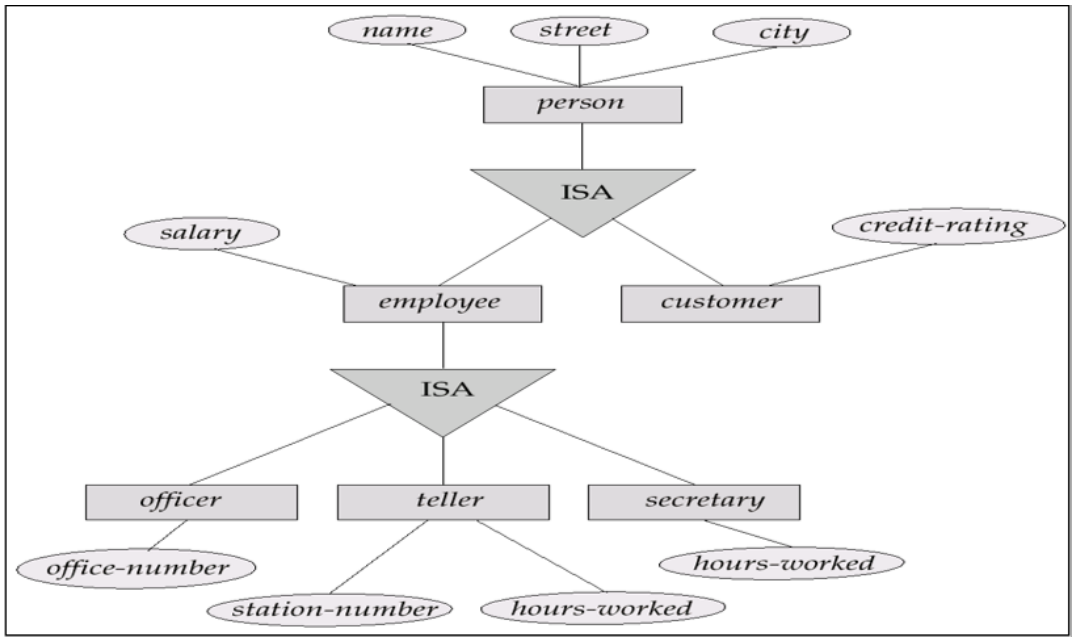
  

- 经常出现在一对多的关系中，在ER图中需要用**双线方框**表示，比如职工和职工家属，职工家属不能脱离于职工存在，所以职工家属就是一个**弱实体集**

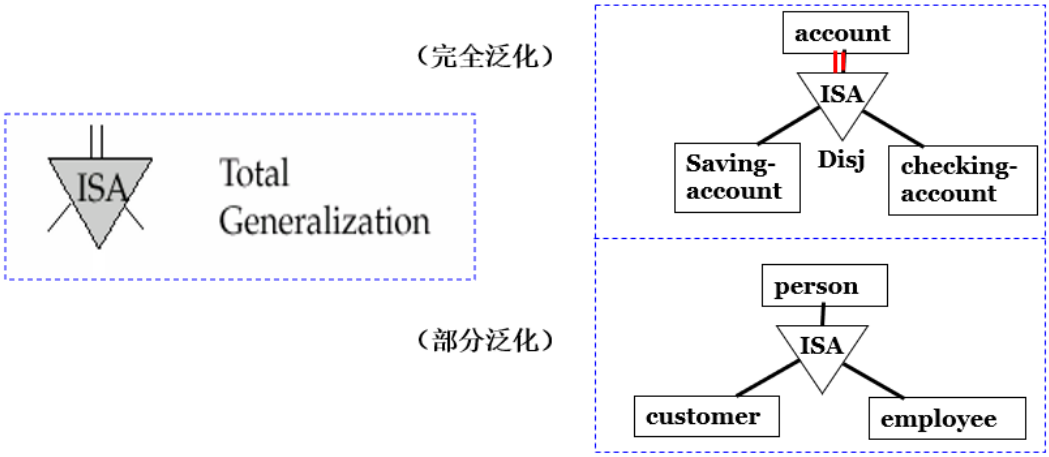
3.1.1 ER设计问题

- 用实体集还是用属性
  - 很难界定
  - 常见错误
    - 一个实体集的主码作为另一个实体集的属性，而不是用联系
    - 将相关实体集的主码属性作为联系集的属性
      - 因为联系集中已经隐含了主码属性
- 用实体集还是用联系集
  - 很难看出哪个更优
  - 业务关系建议实体化
  - 联系只是对应关系
- 二元联系还是n元联系
  - 可以把n元联系转换成二元联系
- Aggregation 聚合
  - 可以把一部分E-R关系聚合成一个Entity进行操作
  - 在ER图中用方框将一些关系集和实体集括起来表示一个聚合后的实体集
  - 消除冗余
- Specialization 特殊化

> 有点oop继承的感觉

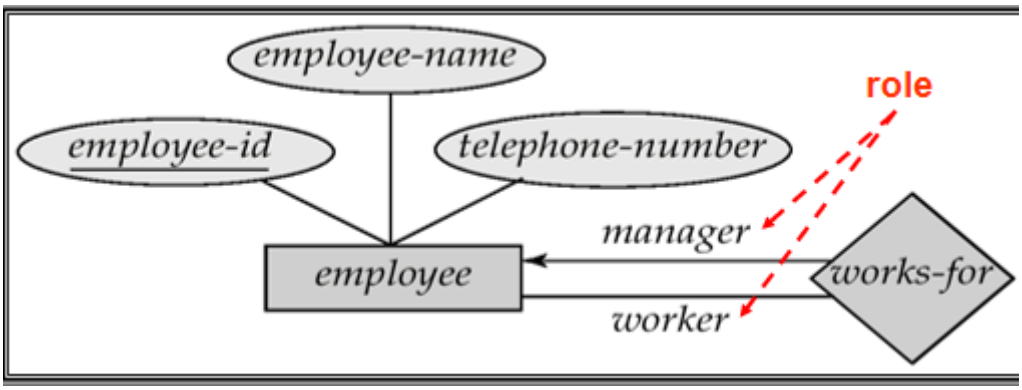
    - 自顶向下的设计过程
    - 特化用从特化实体指向另一方实体的空心箭头来表示——ISA，is a关系
    - Attribute inheritance：
        - overlapping重叠特化
        - disjoint不相交特化
    - 画图的方式就是从上往下画，Entity的内容逐渐细分，但是都继承了上一阶的所有attribute
    - 

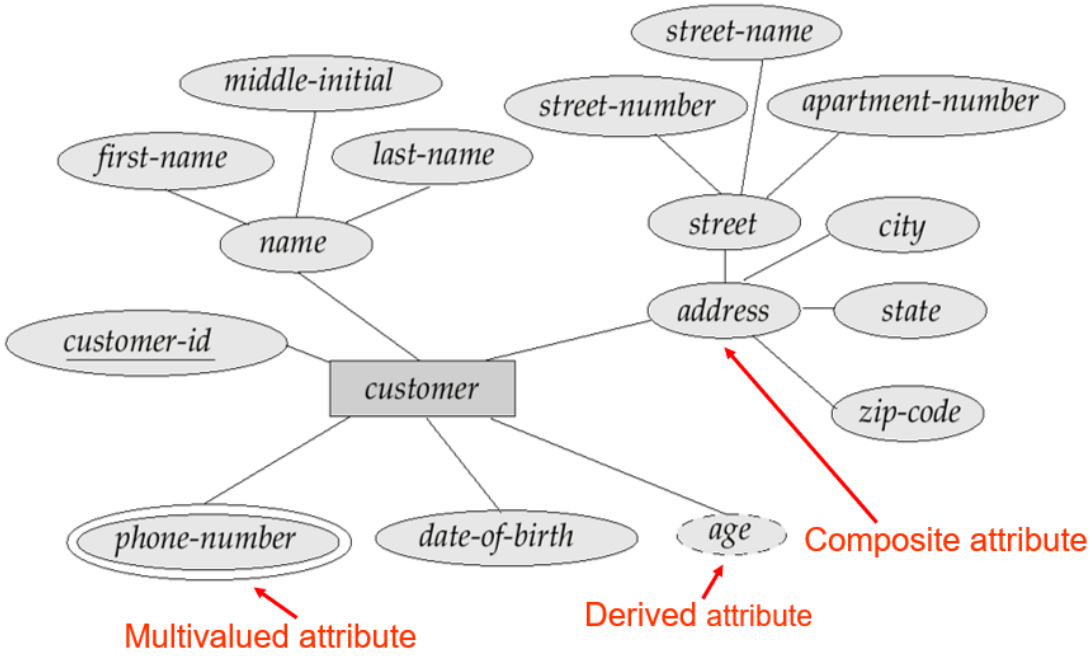
- Generalization 概化
    - 自底向上的设计过程
    - 从下往上，相同属性的下层的内容合成上层的内容
    - 高层与低层实体集也可以分别被称为超类和子类
- Constraint
    - Condition-defined (条件定义的)
    - User-defined
    - Disjoint (不相交)
        - 至多属于一个低层实体集
    - Overlapping (可重叠)
        - 可以重叠多个低层实体集
    - Completeness constraint 完全性约束
        - Total: an entity must belong to one of the lower-level entity sets. 每个高层实体必须属于一个低层实体集
        - Partial: an entity need not belong to one of the lower-level entity sets.允许一些高层实体不属于任何底层实体集
    -



3.2 E-R Diagram

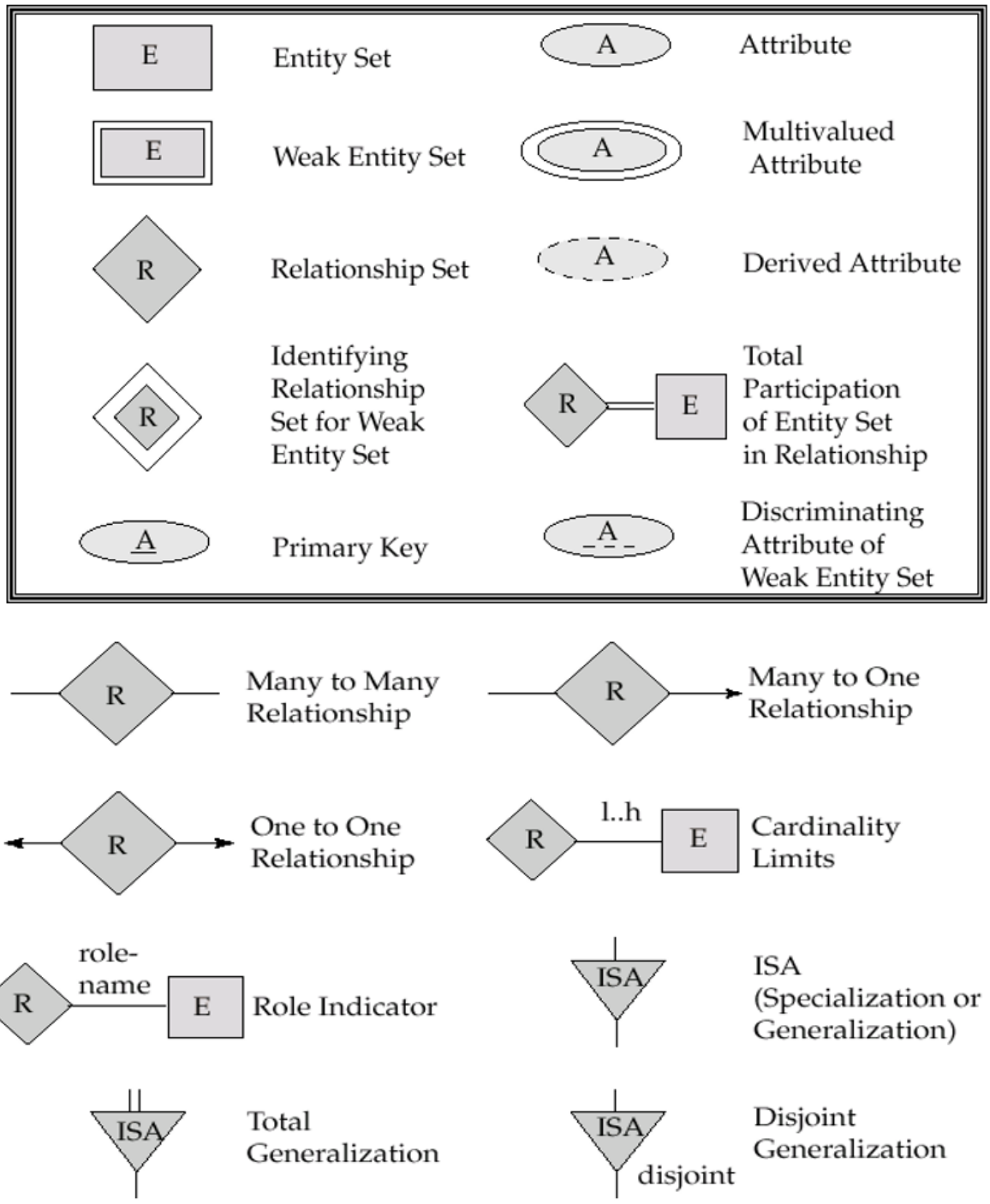
- 矩形是 entity sets.
- 圈是属性
- 菱形是relationship sets.
- Entity sets of a relationship need not be distinct, e.g., Recursive relationship set (自环联系集).





> composite attribute：复合属性

Summary of Symbols Used in E-R Notation



3.3 Normal Form 范式

> 如何判断一个关系型数据库的好坏
>
> - 只有prime key才能决定其他——函数依赖
> - 根据函数依赖，分解成两个或者多个模式，
>   - 回不去——有损分解
>   - 回得去——无损分解
>
> 如果不好，如何改造

3.3.1 数据库设计的目标

- 存储信息时没有不必要的冗余，检索信息的效率高，扁平化
- 这些设计方式通过各种范式(normal form)来实现
- 判断什么数据库是好的数据库
  - 无损分解
- 基于函数依赖、多值依赖

3.3.2 First Normal Form 第一范式

- 原子性atomic： 不能再继续拆分，属性不能再向下拆分
- 第一范式的定义：一个关系模式R的**所有属性都是atomic的**，这个关系模式R就是**第一范式** 1NF
- 存在的问题
  - redundancy 冗余
  - complicates updates 更新数据很复杂
  - null-values---difficult to insert/remove 复杂的操作

> Pitfalls in Relational Database Design
>
> - 将所有的信息用一张大表来表示
>
> 应该分解
>
> - Top-down
> - bottom-up

- Decomposition 分解
  - Lossy Decomposition 有损的分解：不能用分解后的几个关系重建原本的关系
  - Lossless join 无损分解的定义：
    - > 不多不少
    - R 被分解为(R1, R2)并且$R = R_1 \cup R_2$
    - 对于任何关系模式R上的关系r有 $r = \prod_{R_1}(r) \bowtie \prod_{R_2}(r)$
      - iff 至少以下一个成立
        > $R1 \cap R2$是R1或R2的超码
      - $R1 \cap R2 \rightarrow R1$
      - $R1 \cap R2 \rightarrow R2$

> 多个就一个一个来

- Functional Dependency函数依赖
- Multivalued Dependency多值依赖

3.3.3 Functional dependency 函数依赖

- **函数依赖**的定义
  - 对于一个关系模式R，如果$\alpha \subset R$并且$\beta \subset R$则函数依赖$\alpha \to \beta$定义在R上，当且仅当
    - 如果对于R的任意关系r(R) 当其中的任意两个元组t1和t2，如果他们的α属性值相同可以推出他们的β属性值也相同
  - 如果某个属性集A可以决定另一个属性集B的值，就称$A \to B$是一个函数依赖
  - 函数依赖和键的关系：函数依赖实际上是键的概念的一种泛化推广
    - K是关系模式R的**超键**当且仅当 $K \to R$
    - K 是R上的**候选主键**当且仅当 $K \to R$ 并且不存在 $\alpha \subset K, \alpha \to R$，没有多余属性
  - 容易判断一个r是否满足给定的F，但是很难判断F满足schema R
  - 平凡trivial：
    - $\alpha \to \beta$ is trivial if $\beta \subseteq \alpha$
    - 自己或自己的子集决定自己；需要排除
  - 一个平凡的的结论：子集一定对自己函数依赖

3.3.4 闭包

- Closure 闭包
  - 闭包, 对于原始的函数依赖集合F可以推出的所有函数依赖关系产生的集合就是**F的闭包**
  - 符号用F^+表示
  - 函数依赖的性质——Armstrong公理（正确，完备）
    - 自反律reflexity：α的子集一定关于α函数依赖
    - 增补律augmentation：如果$\alpha \to \beta$则有$\lambda\alpha \to \lambda\beta$
    - 传递律transitivity：如果$a \to \beta \wedge \beta \to \gamma$则有$a \to \gamma$
    - 合并律union：如果$\alpha \to \beta \wedge \alpha \to \gamma$则有$\alpha \to \beta\gamma$
    - 分解律decomposition：如果$\alpha \to \beta\gamma$则有$\alpha \to \beta \wedge \alpha \to \gamma$

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

      如果A能推出B，则B是A的闭包

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

    - 属性集闭包的作用
        - 测试是否为主键：如果α的闭包包含了所有属性，则α就是主键
        - 测试函数独立：为了验证$\alpha \to \beta$是否存在只需要验证β是否在α的闭包中
        - 计算F^+：通过每个属性的闭包可以得到整个关系模式的闭包；对任意的$\gamma \subseteq R$，我们找出闭包γ^+；对任意的$S \subseteq \gamma^+$，我们输出一个函数依赖$\gamma \to S$

          > Computing closure of F:
          > R(A,B,C), F={A→B, B→C}
          > computing:
          > A^+=ABC, B^+=BC, C^+=C
          > $(AB)^+$=ABC, $(AC)^+$=ABC, $(BC)^+$=BC, $(ABC)^+$=ABC
          > F⇔{A→ABC, B→BC, C→C, AB→ABC, AC→ABC, BC→BC, ABC→ABC}
          > ⇔{A→A, A→B,A→C, A→AB,A→AC,A→BC,A→ABC
          > B→B,B→C, B→BC,
          > C→C,
          > AB→A, AB→B,AB→C, AB→AB,AB→AC,AB→BC,AB→ABC,
          > AC→A, AC→B,AC→C, AC→AB,AC→AC,AC→BC,AC→ABC,
          > BC→B, BC→C, BC→BC,
          > ABC→A, ABC→B,ABC→C, ABC→AB, ABC→AC, ABC→BC, ABC→ABC}
          > =F^+

- 判定是否为Lossless Join的办法

        - 当且仅当$R_1 \cap R_2 \rightarrow R_1$ 或者$R_1 \cap R_2 \rightarrow R_2$这些函数依赖至少有一个$F^+$中

3.3.5 BCNF/3NF

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

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

- R不属于BCNF，当我们对关系模式R进行分解的时候，我们的目标是
    - 没有冗余，每个关系都是一个good form

- 每一条分解出来的都属于BCNF，如果不是，需要进一步分解
- 无损分解
- 依赖保持

存在至少一个非平凡的函数依赖$\alpha \rightarrow \beta$,其中α不是R的超码，我们在设计中用以下两个模式取代R

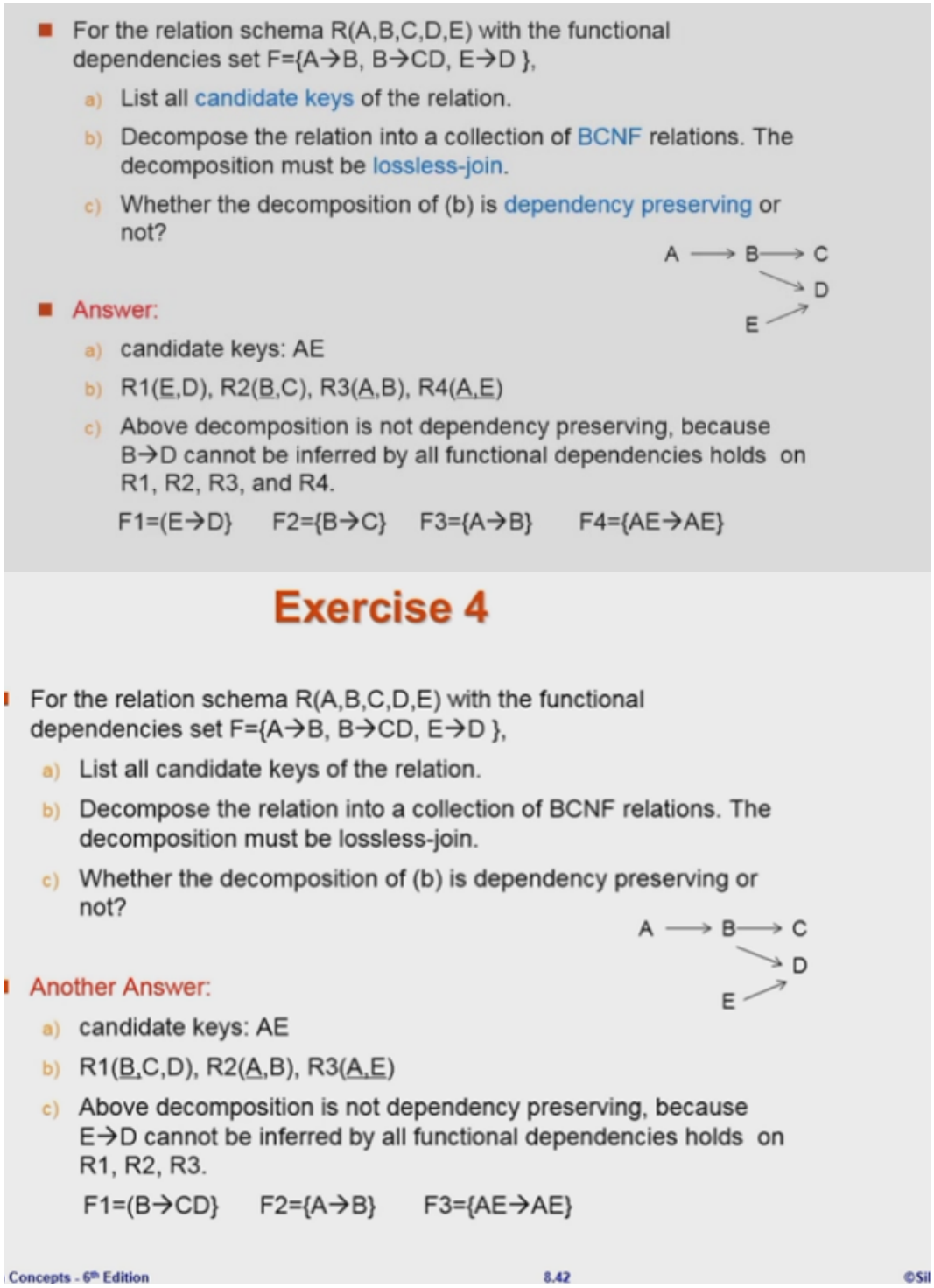
- $(\alpha \cup \beta)$
- $(R-(\beta-\alpha))$

- Denpendency preservation 依赖保持，把R和F的闭包按照关系的对应进行划分

  > 定义：原来关系R上的每一个函数依赖，都可以在分解后的单个的关系上**得到检验**，或者推导出来（方便的检验）

  - 用F_i表示只包含在R_i中出现的元素的函数依赖构成的集合
  - 我们希望的结果是 $(F_1 \cup F_2 \cup \cdots \cup F_n)^+ = F^+$
    - F的每一个函数是否都在这边
    - BCNF的分解一定是有独立性保护的
  - 独立性保护的验证算法
  - 如果最终的结果result包含了所有属性，那么函数依赖$\alpha \rightarrow \beta$ 就是被保护的

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



主要都是根据公式

存在至少一个非平凡的函数依赖$\alpha \rightarrow \beta$,其中α不是R的超码，我们在设计中用以下两个模式取代R

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

- Third normal form 第三范式

> 相较于BCNF，如果硬要依赖保持，要求降低

  - 第三范式的定义：对于函数依赖的闭包F^+中的所有函数依赖$\alpha \rightarrow \beta$ 下面三条至少满足一条

        - $\alpha \rightarrow \beta$ 是平凡的
        - α是关系模式R的超码
        - **每一个β 中的 属性A都包含在一个R的候选码中**
            - 存在多个candidate key，key可能是多字母
            - 候选码最小的超码
    - BCNF一定是3NF，实际上3NF是为了保证依赖保持的BCNF
    - 3NF有冗余，某些情况需要设置一些空值
- 3NF的判定
    - 不需要判断闭包中的所有函数依赖，只需要对已有的F中的所有函数依赖进行判断
    - 用闭包可以检查$\alpha \rightarrow \beta$中的α是不是超键
    - 如果不是，就需要检查β中的每一个属性包含在R的候选键中
- 3NF decomposition algorithm
    - 使用F的正则覆盖
    - 对一条依赖都分解出关系模式
    - 如果一个关系模式里面没有key，需要把某个key拿出来单独放进去

3NF Decomposition Algorithm

Let F_c be a canonical cover for F;
$i := 0$;
for each functional dependency $\alpha \rightarrow \beta$ in F_c do
    {if none of the schemas R_j, $1 \le j \le i$ contains $\alpha\ \beta$
        then begin
            $i := i + 1$;
            $R_i := (\alpha\ \beta)$
        end}
if none of the schemas R_j, $1 \le j \le i$ contains a candidate key for R then
begin
    $i := i + 1$;
    R_i := any candidate key for R;
end
return (R_1, R_2, ..., R_i)

将F_c中的每个$\alpha \rightarrow \beta$分解为子模式$R_i := (\alpha, \beta)$, 从而保证dependency-preserving.

保证至少在一个R_i中存在R的候选码, 从而保证lossless-join.

讨论: 对于多于二个子模式R_i ($i > 2$)的分解, 判别是否无损连接的方法, 其他教材中是用一张i行n列的表来表示. 如果各子模式中函数依赖的相关性使得R中所有的属性都涉及, 则是无损连接分解. 而根据候选码的含义, 候选码必与所有属性相关. 从而二者本质上一致.

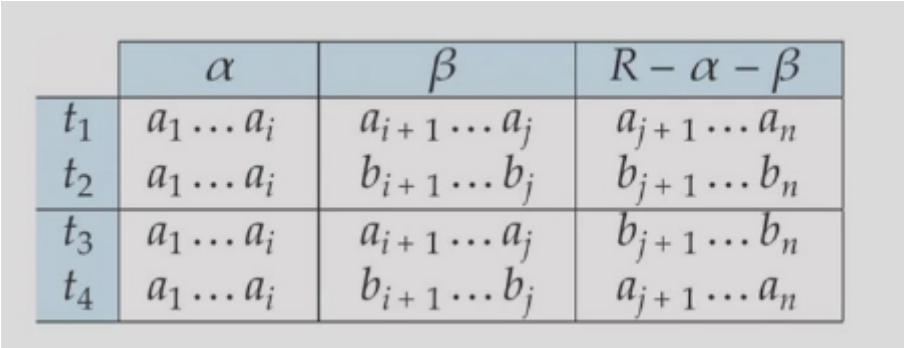
3.3.6 最小覆盖

- Canonical conver 正则覆盖问题
    - 函数依赖关系的最小集合(也就是没有冗余，和F等价可以推导出F+的关系集合)
    - 无关属性Extraneous Attributes：

        - 定义：对于函数依赖集合F中的一个函数依赖$\alpha \to \beta$
            - α中的属性A是**多余**的，如果F逻辑上可以推出$(F - \{\alpha \to \beta\}) \vee \{(\alpha - A) \to \beta\}$
            - β中的属性A是多余的，如果$(F - \{\alpha \to \beta\}) \vee \{\alpha \to (\beta - A)\}$逻辑上可以推出F
                - 更强的函数逻辑上可以推导出更弱的函数
        - 判断$\alpha \to \beta$中的一个属性是不是多余的
            - 测试α中的属性A是否为多余的
                - 计算$(\alpha - A)^+$
                - 检查结果中是否包含β，如果有就说明A是多余的
            - 测试β中的属性A是否为多余的
                - 只用$(F - \{\alpha \to \beta\}) \vee \{\alpha \to (\beta - A)\}$中优的依赖关系计算$\alpha^+$
                - 如果结果包含A，就说明A是多余的
    - 最小覆盖F_c的定义
        - 和F可以互相从逻辑上推导出，并且最小覆盖中没有多余的信息
        - 性质：
        - F_c中任何函数依赖都不含无关属性
        - 最小覆盖中的每个函数依赖中左边的内容都是unique的
        - 如何计算最小覆盖：PPT-8的53页有一个例子
            - 先令Fc=F
            - 用Union rule将Fc中所有满足$\alpha \to \beta_1 \wedge \alpha \to \beta_2$的函数依赖替换为$\alpha \to \beta_1\beta_2$
            - 找到Fc中的一个函数依赖去掉里面重复的属性
            - 重复2，3两个步骤直到Fc不再变化
- Multivalued denpendency

> 多值依赖不同于函数依赖，他要求某种形式的元祖存在于关系中，函数依赖有时被称为**相等**产生依赖，多值依赖被称为**元祖**产生依赖。

    - 多值依赖 a->->b，记作D；闭包记为D+，是由D逻辑蕴涵的所有函数依赖和多值依赖的集合。

- 

	α	β	$R-\alpha-\beta$
t_1	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
t_2	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
t_3	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
t_4	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$

  a和b之间的联系独立于a和$R-b$之间的联系；如果R上的所有关系都满足多值依赖，则称为是平凡的多值依赖
  - 若a->b，则a->->b。换句话说，每一个函数依赖也是一个多值依赖
  - 若a->->b，则a->->R-a-b
- Fourth Normal Form
  - 对于D+中的所有a->->b多值依赖有①是平凡的多值依赖或者②a是一个超键
  - 4NF一定是BCNF
  - 4NF分解
    - ## 4NF Decomposition Algorithm

      result := {*R*};
      done := false;
      compute D^+;
      Let D_i denote the restriction of D^+ to R_i
        while (not *done*)
          if (there is a schema R_i in *result* that is not in 4NF) then
            begin
              let $\alpha \rightarrow\rightarrow \beta$ be a nontrivial multivalued dependency that
              holds on R_i such that $\alpha \rightarrow R_i$ is not in D_i, and $\alpha \cap \beta = \phi$;
                result := (*result* – R_i) ∪ (α, β) ∪ (R_i – β);
            end
          else *done*:= true;

      R_{i1} R_{i2}

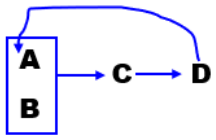
      Note: each R_i is in 4NF, and decomposition is lossless-join.

> 但有时候不需要那么规范化，可以有一些数据的冗余

Example 2

- For a relation schema $R(A, B, C, D)$ with $F = \{AB \rightarrow C, C \rightarrow D, D \rightarrow A\}$.
  - 1) List all the candidate keys for the relation schema R.
  - 2) Decompose the relation schema R into a collection of BCNF relation schemas.
  - 3) Explain whether the decomposition of 2) is dependency preserving.

- Answer:



1): $(AB)^+ = (ABCD) \supseteq R$, $(BC)^+ = (ABCD) \supseteq R$,
$D \rightarrow A$, $BD \rightarrow AB$; $AB \rightarrow C$; $\therefore (BD)^+ = (ABCD) \supseteq R$,
$\therefore$ AB, BC, BD are candidate keys, and R is not in BCNF.

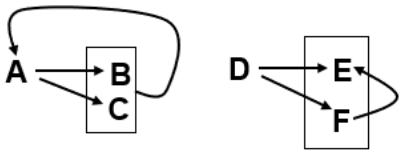
2): ⇨ $R_1(C, D)$; $R_2(A, B, C)$,
(R_1 is in BCNF, R_2 is not BCNF, $\because C \rightarrow D$, $D \rightarrow A$, $\therefore C \rightarrow A$, C is not key of R_2);
$R_{21}(A, C)$, $R_{22}(B, C)$, R_{21} is in BCNF, R_{22} is in BCNF.

3): $D \rightarrow A$, $AB \rightarrow C$ are not preserved.

Example 3

- $R = (A, B, C, D, E, F)$, $F = \{A \rightarrow B, A \rightarrow C, BC \rightarrow A, D \rightarrow EF, F \rightarrow E\}$
  - 1) Find all candidate keys.
  - 2) Whether R is in BCNF or 3NF?
  - 3) If it is not in BCNF, decompose R into a set of BCNF relations. Explain that your decomposition is lossless-join.
  - 4) Whether the decomposition of 3) is dependency preserving or not? Why?

- Answer:



1) Candidate keys: AD, BCD

2) $\because F \rightarrow E$, F is not a key, E is not in key, $\therefore$ not BCNF, not 3NF.

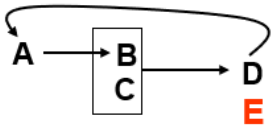
3) $R_1 = (A, B, C)$, $R_2 = (A, D, E, F)$. (由$A \rightarrow BC$), $R_{21} = (D, E, F)$, $R_{22} = (A, D)$, but R_{21} is not BCNF, $\because F \rightarrow E$, (AD is key) $R_{211} = (F, E)$, $R_{212} = (D, F)$

4) $D \rightarrow E$ is not preserved.

方法 2: $R_1 = (B, C, A)$, $R_2 = (B, C, D, E, F)$; $R_{21} = (F, E)$, $R_{22} = (B, C, D, F)$, $R_{221} = (D, F)$, $R_{222} = (B, C, D)$. Thus, $D \rightarrow E$ is not preserved.

Example 4

- $R = (A, B, C, D, E)$, $F = \{A \rightarrow B, BC \rightarrow D, D \rightarrow A\}$
  - 1) Find all candidate keys.
  - 2) Whether R is in BCNF or 3NF or neither?
  - 3) If it is not in BCNF, decompose R into a set of BCNF relations. Explain that your decomposition is lossless-join.
  - 4) Whether the decomposition of 3) is dependency preserving or not? Why?
- Answer:



1) Candidate keys: *ACE*, *BCE*, *CDE*

2) ∵Every right attribute in F is in key. ∴ R is 3NF.

3) $R_1 = (AB)$, $R_2 = (ACDE)$, $R_{21} = (AD)$, $R_{22} = (CDE)$

3′) $R_1 = (AB)$, $R_2 = (ACDE)$, $R_{21} = (ACD)$, $R_{22} = (ACE)$, $R_{211} = (AD)$, $R_{212} = (CD)$

3″) … …

4) $BC \rightarrow D$ is not preserved.

第四部分: 数据库设计理论

4.1 存储和文件结构

- 存储的结构 storage hierarchy
  - primary主存储器
    - 快而易失，常见的有主存和cache
    - cache的存取效率最高，但是costly，主存访问快但是对于数据库而言空间太小，易失
  - secondary 二级存储器
    - 不容易丢失，访问较快，又叫在线存储
    - 常见的是闪存flash和磁盘magnetic disk
    - 数据库大多数把数据存在磁盘上（越来越多的在闪存上）
    - SSD，内部是flash
  - tertiary三级存储器
    - 不容易丢失，访问慢，但是容量大而cheap，离线存储
    - 磁带，光存储器
  - 总体的存储架构：cache--主存--闪存--磁盘--光盘--磁带

4.1.1 磁盘 Magnetic Disks

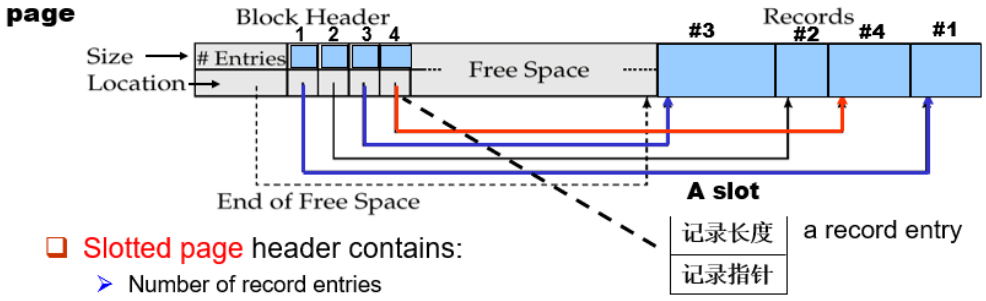
- 组成结构
  - read-write head 读写头
    - 和磁盘表面靠得很近

        - 用于读写磁盘中的文件信息
    - tracks 磁道, 由磁盘表面划分，每个硬盘大概有50k到100k个磁道
        - sectors 扇区，由磁道划分而成
            - 扇区是数据读写的**最小单位**
            - 每个扇区的大小是512字节，每个磁道有500-1000个扇区
    - 磁盘控制器：计算机系统和磁盘之间的接口
    - Disk subsystem 磁盘子系统：由disk controller操纵若干个磁盘组成的字系统
- 磁盘的性能评价标准
    - access time： 访问时间，包括
        - seek time：读写头的arm正确找到track的时间，平均的seek time是最坏情况的一半
        - rotational latency：旋转造成的延迟，平均时间是最坏的一半
    - data-transfer rate 数据从磁盘读写的速度
    - MTTF：出现failure之前的平均运行时间
    - IOPS：每秒进行I/O操作的次数，50~200
        - 瓶颈
- 磁盘访问的优化
    - block： 一个磁道中的**若干连续扇区**组成的序列
        - 是一个逻辑单元，块大小在512字节到几KB之间
        - 数据局部性
    - buffering：暂时存储在内存缓冲区；类似cache的概念
    - read-ahead：相同磁道的连续块也会被读入内存缓冲区
    - disk-arm-scheduling：从最内到最外，然后再从外到最内，对有访问请求的磁道停下
    - file organization：通过按照访问数据最接近的方式来组织block优化访问时间
    - nonvolatile write buffer：非易失性写缓冲区NV-RAM，先写到RAM里，等到磁盘没有其他请求或者RAM满了再写到磁盘里
    - log disk日志磁盘：顺序执行，先

4.1.2 File organization 文件组织

- 数据库存储在一系列的文件中，每个文件是一系列的记录，每条记录包含一系列的fields

- 每个文件被划分为固定长度的block，block是数据存取/存储空间分配的基本单位，大多数数据库默认使用4KB-8KB的块
- 一个block有多条记录，在传统的数据库中
  - 记录的长度不能超过block
  - 每条记录包含在单个块中
    - 余下的字节不用了
  - 每条记录一定都是完整的
  - 删除记录比较困难
    - Free List 用链表的形式来存储删除records
    - 直接覆盖
- Variable-length records 变长记录
  - 典型的变长记录
    - 属性按照顺序存储
    - 变长的变量用offset+data的形式存储，offset是开始的位置，data是长度；空值用null-value bitmap存储，空的话对应的位置1
  - slotted page结构，它的header包含



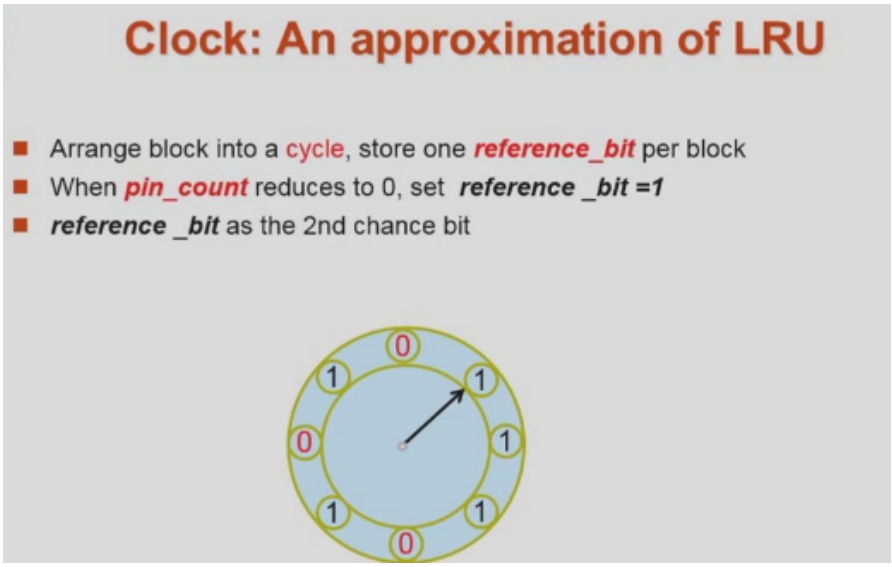
    - 记录的总数
    - block中的空闲区域的end
    - 每条记录所在的位置和大小
    - 插入：在空闲空间的尾部分配空间，插入
    - 删除：空间被释放重用，条目被设置成被删除状态
- 文件中记录的组织方式
  - heap：没有顺序，有空就插
    - 有二级的free-space map，每几个为一组，标记这个组里的free-space
  - sequential：顺序
    - 删除用指针链表
    - 插入先插到溢出块当中，用链表链接，当插入很多的时候，就重组，使得他再一次在物理上顺序存放
  - hashing

        - B+ - Tree ： 叶子直接放记录；存在一些问题
            - 可能改变记录的位置，即使记录没有更新
                - 辅助索引中，不存储指向被索引的记录的指针，而是存储主索引搜索码属性的值；叶节点分裂导致的记录重定位就不需要对辅助索引进行更新
            - 记录字符串可能变长
            - 字符串可能会很长
                - 前缀压缩，前缀树，中间只需要分出路径即可，叶子是完整的
            - 批量加载bulk loading/集体插入：从底向上构建b+树，先自己排好序，再操作
        - multi-table clustering file organization：几个不同关系的记录存储在同一个文件里
- 数据字典存储

    > 关于数据的数据：元数据
    >
    > 关于关系的关系模式和其他元数据存储称为 **数据字典** 或 **系统目录**

    - 关系的名字
    - 权限管理
    - 用户信息
    - 统计信息
    - 存储方式
- 存储缓冲区的管理
    - 通过将数据放到主存中来提高访问效率
        - buffer manager：用于管理缓冲区中的内存分配
            - 当需要从磁盘读取block的时候，数据库会调用buffer mananger的功能
            - 如果block已经在buffer中了，就直接返回这个block的地址
            - 如果不在，则buffer manager 会动态分配buffer中的内存给block，并且可能会覆盖别的block，然后将磁盘中block中的内容写入buffer中
                - 如果满了，又miss，就涉及到buffer的**替换算法**LRU strategy 即替换掉最近使用频率最低的block
                - LRU：最近最少使用替换策略，队列

    - clock算法：



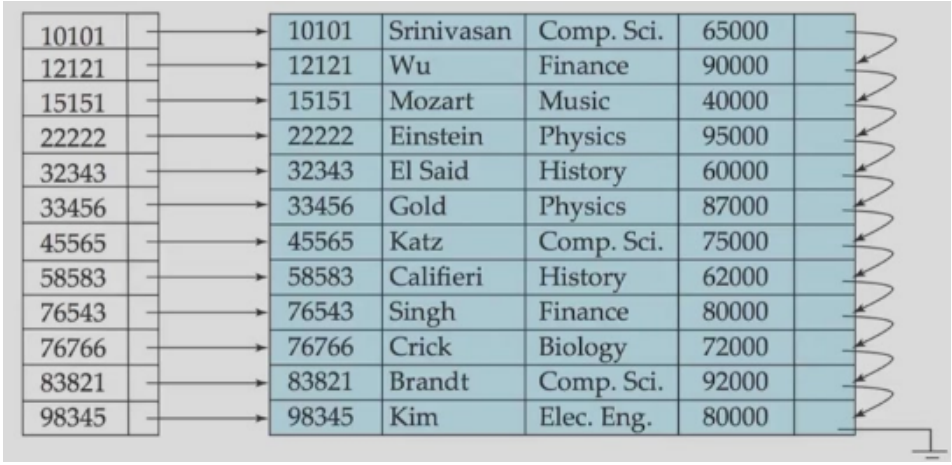
    - pinned block 内存中的不允许写回磁盘的block，表示正在处理事务或者处于恢复接断
    - forced output of block：强制写出

4.2 B+树索引

4.2.1 索引

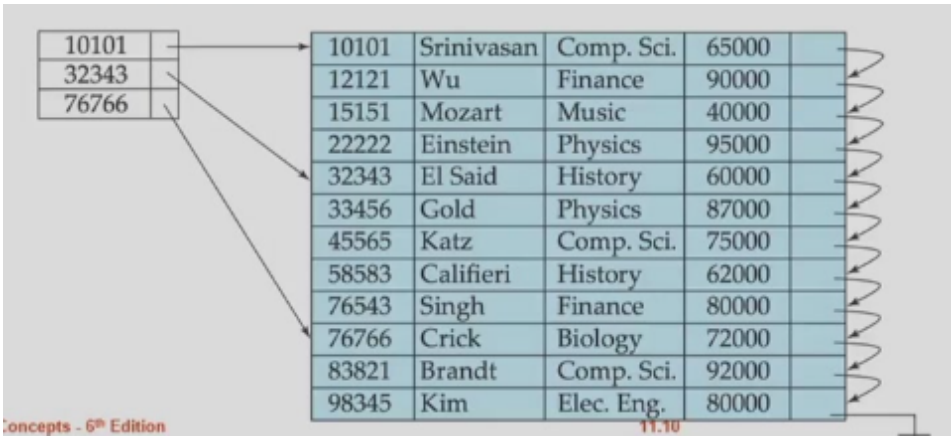
- 数据库系统中引入索引机制，用于加快查询和访问需要的数据
    - search key 通过一个属性值查找一系列属性值，用于文件中查询
    - Index file **索引文件**包含一系列的search key和pointer(两者的组合被称为index entry)，查询方式是通过search key在index file中查询data的地址(pointer)，然后再从data file中查询数据
        - 两种search key的排序方式：ordered indices，hash indices
        - ordered index 顺序索引
            - index entry按照search key的值来进行排列
            - primary key 指定文件顺序的索引
            - secondary key 次关键字，**搜索码指定的顺序与文件中记录的物理顺序不一样**
        - 索引的不同方式

- Dense index 密集的索引：每一条记录都有对应的索引



10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

- Sperse index 稀疏的索引：只为某些值建立索引项



10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

  - 需要的空间和插入删除新索引的开销较小，但是比密集的索引要慢
- Secondary indice索引通过一个大的bucket来寻找所指向的地方
- Multilevel index 多级索引，分为outer index和inner index

4.2.2 B+树索引

- B+树文件索引
  - 通过B+树的索引方式来寻找文件中数据的地址，B+树的定义和ads中的B+树基本相同，
    - 树的非叶节点由指向儿子的指针和search-key相间组合而成

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

    - 最多有$n-1$个key，n个指针；最小有$\lceil (n-1)/2 \rceil$
    - n-1个key表明了n个块——>非叶节点最多可以有n个儿子
    - 两个search-key之间的指针指向的数据的值在这两个search-key之间
  - B+树上的查询的时间复杂度是$logN$级别，N是search key的总个数
    - 查询的路径长度：不会超过$\log_{n/2}(K/2)$ + 1其中K是B+树中的索引的个数(即规模N)

        - B+树的一个节点的大小和一个磁盘区块一样大(往往是4KB)而在n的规模一般在100左右
- B+树的更新：插入和删除
    - 插入的算法：先找到该插入的位置直接插入，如果当前的节点数量超过了阶数M则拆成两个部分，并向上更新索引
    - 删除的算法: 直接把要删除的节点删除，然后把没有索引key了的非叶节点删除，从旁边找一个叶节点来合并出新的非叶节点
- B+树的相关计算
    - 高度的估计：
        - B+树高度最小的情况：所有的叶节点都满，此时的$h = \lceil \log_N(M) \rceil$
        - 最大的情况，所有的叶节点都半满，此时的$h = \lfloor \log_{\lceil N/2 \rceil}(\frac{M}{2}) \rfloor + 1$
    - size大小的估计：也是两种极端情况
        - 除n向上取整
        - 除$\lceil n/2 \rceil$ 向上取整
    - 500 + 167+56+19+7+3+1
      753

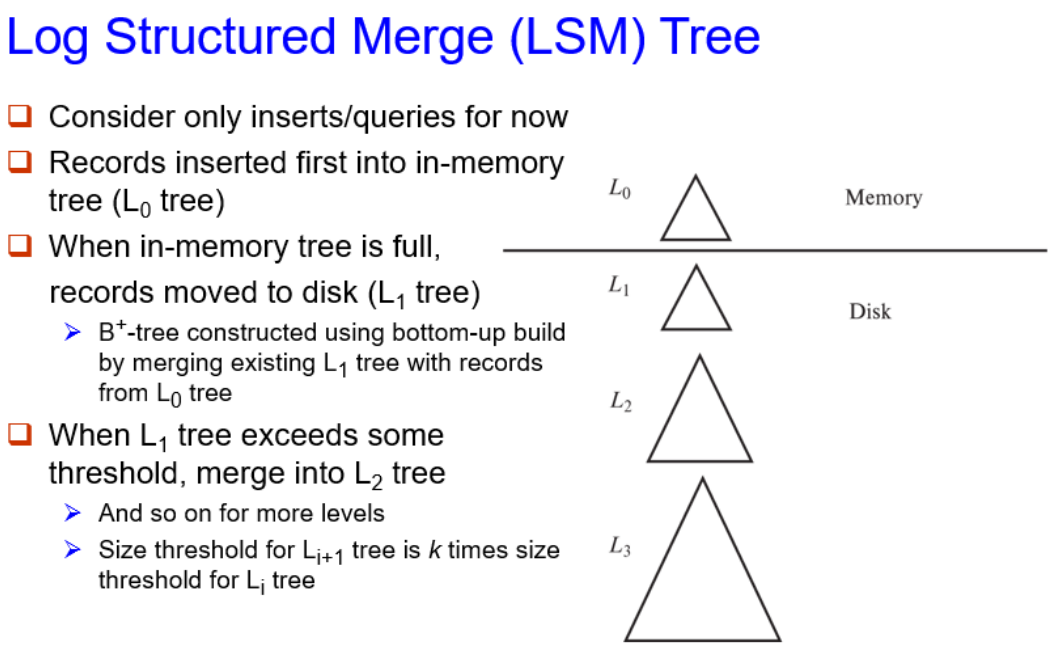
$$\text{size} \geq \left\lceil \frac{1000}{2} \right\rceil + \left\lceil \left\lceil \frac{1000}{2} \right\rceil /3 \right\rceil + \left\lceil \left\lceil \left\lceil \frac{1000}{2} \right\rceil /3 \right\rceil /3 \right\rceil + \cdots + 1 = 755$$

$$\text{size} \leq 1000 + \lceil 1000/2 \rceil + \lceil \lceil 1000/2 \rceil /2 \rceil + \cdots + 1 = 2001$$
$$755 \leq \text{size} \leq 2001$$

> - *person*(*pid* **char**(18), *name* **char**(8), *age* **smallint**,
> - address **char(40),** **primary key** (*pid*));
> - *Block size : 4K*
> - *1000000 persons*
> - *Records per block* = 4096/(18+8+2+40) =60.235 → 60
>
> blocks for storing 1M persons= ⌈ 1000000/60 ⌉=16667
> B+ tree n(fan-out) = (4096-4)/(18+4) +1 = 187
> ⌈n/2⌉= 94 , inner pointers : 94~187
> ⌈n-1/2⌉=93, leaf values : 93~186
> 2 levels: min=2*93 =186 max= 187*186= 34,782
> **3 levels**: min=2*94*93 =17484 max= 187*187*186=6,504,234
> 4 levels: min=2*94*94*93=1,643,496 max= 187*187*187*186=
> 1,216,291,758

> fan-out计算：块的大小/（search-key+pointer）的大小

4.2.3 文件索引

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

  - 先写进一个树，当满了再merge

4.2.4 总结：存储结构和B+树的计算

- 记录的存储：
  - 数据库的记录在block中存储，一个block中有大量的记录存储，有线性存储的，也有使用B+树索引的
  - 线性存储的记录：

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

4.3 查询处理 QueryProcess

- 查询处理的基本步骤
    - Parsing and translation 语法解析和翻译
    - Optimization 优化
        - 一种SQL查询可能对应了多种等价的关系代数表达式
        - 可以通过估计每种方式的cost来评判方法的好坏
        - 查询优化会选择最节约的方式进行查询
    - Evaluation
- Query cost的计算
    - 主要的cost来源：disk access
        - seeks
        - block read
        - block written
    - cost计算的方式：在B个blocks中查询S次所消耗的时间=B*转移到一个block的时间+S*一次查询的时间：$B \times t_T + S \times t_S$ 其中t_T 表示一次block transfer的时间
        - cost依赖于主存中**缓冲区的大小**：更多的内存可以减少disk access
        - 通常考虑最坏的情况：只提供最少的内存来完成查询工作

4.3.1 select的cost估计

- Select 操作的cost计算
    - Algorithm1:线性搜索，查询每个block判断是否满足查询条件
        - Cost = br * block transfers + 1 seek，其中b_r 是关系r中**存储了记录的block的数量**

        - 如果通过键来搜索，在找到的时候就停止，则 average cost = (br /2) * block transfers + 1 seek
        - 二分搜索此时不起作用，因为数据不是连续存储的
    - Index scan--使用索引进行搜索
    - Algorithm2: primary index，equality on key，搜索一条记录
        - $cost = (h_i + 1) \times (t_T + t_S)$ --- hi是索引的高度
    - Algorithm3: primary index, equality on non-key 需要搜索多条记录，非码
        - 想要的结果会存储在连续的block中(因为有主索引)
    - $cost = h_i(t_T + t_S) + t_S + t_T * b$ 其中b表示包含匹配记录的block总数
        - Seek时间不变，block transfer变多了，因为在多个块内有值
        - 当使用B+树作为索引时可以节约一次seek的时间，$cost = h_i(t_T + t_S) + t_T * b$
- 算法4：Secondary index
    - 用候选主键作为索引检索单条记录 $cost = (h_i + 1) \times (t_T + t_S)$
    - 用候选主键检索了n条记录(不一定在同一个block上面)

      $Cost = (h_i + n) \times (t_T + t_S)$ 有时候会非常耗时
- 算法5：B+树主索引，比较$cost = h_i(t_T + t_S) + t_T * b$
- 算法6：B+树辅助索引，比较$Cost = (h_i + n) \times (t_T + t_S)$
- 算法7：利用一个索引的合取选择
    - 我理解的是先选择再合取
- 算法8：使用组合索引的合取选择
- 算法9：通过标识符的交实现合取选择
    - 我理解的是先合取再选择
- 算法10：通过标识符的并实现析取选择

4.3.2 sort和join的cost估计

- Sort：**external sort-merge** 其实类似于ads里面的外部归并排序
    - M表示内存的大小，内存缓冲区中可以用于排序的块数，b_r表示block的数量
        - 基本步骤如下
            - create sorted runs
            - merge the runs
        - 需要的merge pass归并趟数总数 $[\log_{M-1}(b_r/M)]$
        - 创建和每次run过程中的disk access数量 $2 * b_r$

        - 外部排序中总的disk access次数 $(2\lceil \log_{M-1}(b_r/M) \rceil + 1)b_r$
            - 因为少了最后的一次
- Seek：
    - 如果每次从一个归并段读取b_b块数据，把每一趟归并需要作 2*$\lceil b_r/b_b \rceil$次，总次数为$2\lceil b_r/M \rceil + \lceil b_r/b_b \rceil (2\lceil log_{\lfloor M/b_b \rfloor -1}(b_r/M) \rceil - 1)$
    - 会有极值
- Join 操作的cost估计
    - nested-loop join
        - 计算theta-join表达式：$r \bowtie_\theta s$ 算法的伪代码如下

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

        - n_r是r中的元组数，b_r是r中元组的磁盘块数，b_s是s中元组的磁盘块数，n_s是s中的元组数
        - block transfer次数: $n_r \times b_s + b_r$
        - seeks的次数 $n_r + b_r$
    - block nested-loop join $r \bowtie_\theta s$

```
for each block Br of r do begin
    for each block Bs of s do begin
        for each tuple tr in Br  do begin
            for each tuple ts in Bs do begin
Check if (tr,ts) satisfy the join condition
                if they do, add tr • ts to the result.
```

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

        - seek次数= $\frac{2b_r}{M-2}$
    - Index nested-loop join
        - 索引一定程度上可以代替file scan
        - $cost = b_r(t_T + t_s) + c \times n_r$ 其中c表示遍历索引和找到所有匹配的s中的tuple所消耗的时间，可以用**一次s上的单个selection来估计s的值**
    - Merge-Join
        - 只能在natural-join和equal-join中使用
        - 假设为每个关系分配b_b个缓冲块
        - block transfer的次数=$b_r + b_s$，seek的次数=$\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil$
    - Hash join：使用hash函数进行join
        - *h* maps JoinAttrs values to {0, 1, ..., n_h}，将两个关系进行比较和同类型的匹配
        - cost of hash-join
            - 不需要递归：
                - block transfer: $3(b_r + b_s) + 4n_h$
                    - partition: 读 $b_r + b_s$ blocks 写 $(b_r + b_s) + 2n_h$ blocks
                    - join: 读$(b_r + b_s) + 2n_h$
                - seeks: $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil) + 2n_h$
                - 如果所有东西都能放进主存里，则$n_h = 0$ 并且不需要partition
            - 需要递归：
                - Block transfer: $cost = 2(b_r + b_s)\lceil \log_{M-1}(b_s) - 1 \rceil) + b_r + b_s$
                - Seek:$2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil)\lceil log_{M-1}(b_s) - 1 \rceil$
                - 每一个划分小于等于M
- 其他运算
    - 去除重复
    - 投影
    - 集合运算
        - hash改造一下
    - 外连接
- Evaluation of Expression 表达式求值
    - **Materialization** 实体化
        - 依次进行表达式的计算，构建前缀树递归进行
        - 每次结果写回磁盘以备后用

- Pipelining 流水线
  - evaluate several operations simultaneously , passing the results of one operation on to the next.
  - 同时计算多个操作，一个运算的结果传递给下一个
  - 需求驱动的流水线
    - 有需求请求，再计算返回
  - 生产者驱动的流水线

4.4 查询优化 Query Optimization

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

4.4.1 等价关系代数表达式

- Equivalent Expressions 等价的关系代数表达式
  - **evaluation plan**：类似于算术表达式的前缀树，表示了每部操作进行的过程
  - Cost-based optimization基于cost的优化
    - 基本步骤
      - 用运算法则找到逻辑上**等价的表达式**
        - 等价变换规则
      - 注释结果表达式来获得查询计划
      - 选择cost最低的表达式
    - cost的估算
      - 统计信息量的大小，比如tuples的数量，一个属性不同取值的个数
      - 中间结果的数量，用于复杂表达式的优化（基数估计Cardinality Estimation）
      - 算法的消耗
  - **等价表达式的规则**

4.4.2 关系表达式的转换

> 等价规则

- 合取选择和选两次等价：$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$
- 选择两次的顺序可以交换；$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$
- 嵌套的投影只需要看最外层的：$\prod_{L_1}(\prod_{L_2}(\ldots(E))) = \prod_{L_1}(E)$
- 选择可以变成笛卡尔积和theta join
  - $\sigma_\theta(E_1 \times E_2) = E_1 \bowtie_\theta E_2$
    - $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$
- Theta-join和自然连接可以改变连接的两张表的顺序：
  $E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$
- 自然连接满足结合律：$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$
  - Theta-join的结合规则

  $(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_2 \wedge \theta_3} E_3 = E_1 \bowtie_{\theta_1 \wedge \theta_3} (E_2 \bowtie_{\theta_2} E_3)$
- 选择操作的优化，选择运算对于theta join运算具有分配律
  - 当θ_1 中的属性都只出现在E1中的时候：
    $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = \sigma_{\theta_1}(E_1) \bowtie_{\theta_2} E_2$
  - 当θ_1, θ_2 分别只包含E1,E2中的属性时：
    $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_\theta E_2) = \sigma_{\theta_1}(E_1) \bowtie \sigma_{\theta_2}(E_2)$

  > 选择操作先做，压到叶子

- 投影操作和Theta-join的混合运算，投影对theta join具有分配律
  - 当θ只包含$L_1 \vee L_2$ 中的属性的时候：
    $\prod_{L_1 \vee L_2}(E_1 \bowtie_\theta E_2) = (\prod_{L_1}(E_1)) \bowtie_\theta (\prod_{L_2}(E_2))$
  - 8. The projection operation distributes（分配） over the theta join operation as follows:

    (a) if θ involves only attributes from $L_1 \cup L_2$:

    $$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

    (b) Consider a join $E_1 \bowtie_\theta E_2$.

    - Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.
    - Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
    - let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$.

    $$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

- 集合运算中的交运算和并运算满足交换律和结合律
- 选择操作中有集合的运算时满足分配律(比如进行差运算再选择等价于分别选择再差运算)
  - $\sigma_P(E_1 - E_2) = \sigma_P(E_1) - \sigma_P(E_2)$
  - 把-换成∪或者∩都可以
- 投影操作中有并运算时满足分配律
  - $\prod_L(E_1 \cup E_2) = (\prod_L(E_1)) \cup (\prod_L(E_2))$
- **选择**尽早的选，以减少大小；**投影**尽早的投影，以减少大小
- Join的顺序优化：当有若干张表需要join的时候，**先从join后数据量最小的开始**
- 可以通过共享相同的子表达式来减少表达式转化时的空间消耗，通过动态规划来减少时间消耗

4.4.3 cost的估计

- 基本的变量定义
  - n_r 表示关系r中元组的数量(也就是关系r的size)
  - b_r 包含r中元组的block数量
  - l_r r中一个元组的字节数
  - f_r block factor of r 比如可以选取一个block能容纳的r中元组的平均数量
  - V(A, r) 关系r中属性A可能取到的不同的值的数量，等于$\prod_A(r))$
  - 当关系r中的元组都存储在一个文件中的时候 $b_r = \frac{n_r}{f_r}$
- 直方图histogram
- 选择的估计
  - 从r中选择A属性=x的$cost = \frac{n_r}{V(A,r)}$
  - 选择A属性小于x的cost
    - $cost = 0$ if x < min(A,r)
    - $cost = n_r \cdot \frac{v-min(A,r)}{max(A,r)-min(A,r)}$
  - 选择A属性大于x，和上面的表达式是对称的
- complex selection 多重选择
  - 假设s_i是满足条件θ_i的元组的个数
  - conjunction $cost = n_r \times \frac{s_1 \times s_2 \times \ldots \times s_n}{n_r^n}$
    - 独立分布
  - disjunction $cost = n_r \times (1 - (1 - \frac{s_1}{n_r}) \times \ldots \times (1 - \frac{s_n}{n_r}))$
    - 某一个条件成立
  - negation $cost = n_r - size(\delta_\theta(r))$

- join 的估计
  - 笛卡尔积的情况下，关系R,S的join最终元组的个数为$n_r \times n_s$
  - 如果R∩S为空，则自然连接的结果和笛卡尔积的结果相同
  - 如果非空，且R∩S是R的key，则R,S的自然连接最终结果中的元组个数不会超过s中元组的数目
  - 如果R∩S的结果是S参照R的外键，则最后的元组数和s中的元组数相同
  - 一般情况 自然连接的最终结果的size估计值为$\frac{n_r \times n_s}{max(V(A,r),V(A,s))}$
- 其他操作的估计
  - 投影的size=V(A,r)
  - 聚合操作的size=V(A,r)
  - 集合操作：
  - 外部连接：
    - 左外连接的size = 自然连接的size + r的size
    - 右外连接的size = 自然连接的size + s的size
    - 全连接的size = 自然连接的size + r的size + s 的size
  - 不同值个数的估计
- 基于cost的join顺序优化
  - n个关系进行自然连接有$\frac{(2n-2)!}{(n-1)!}$种不同的join顺序
  - 找到最合适的join-tree的办法：递归地尝试,局部搜索的办法——>动态规划
    - 时间复杂度$O(3^n)$，空间复杂度$O(2^n)$
    - 随机分成$S-S1$和$S1$，然后比较是否最佳
  - Left Deep Join Trees左倾树，当结合方式只考虑左倾树的时候，找到最优解的时间复杂度是$O(n2^n)$,空间复杂度$O(2^n)$
    - 分成1和$n-1$
- **Heuristic Optimization** 启发式的优化
  - 尽早进行selection
  - 尽早进行projection
  - 选择最严格的selection和operations操作
- 用于查询优化的结构
  - pipelined evaluation plan
  - optimization cost budget
  - plan catching

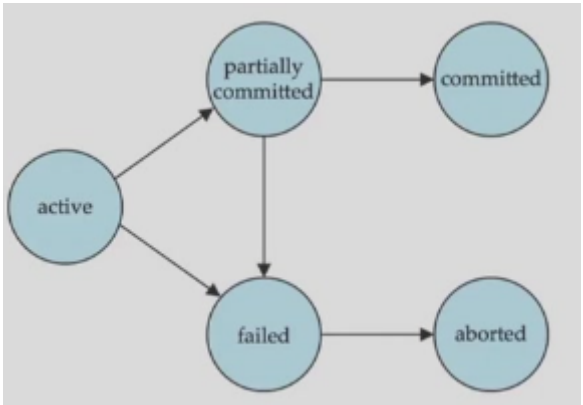
第五部分：事务处理

- 和操作系统关系比较密切

5.1 事务和并发控制

5.1.1 基本的概念

- 事务的概念
  - **事务**时**程序执行**的基本单位，会引起一些数据项的更新，需要解决的两个问题：
    - 数据库系统的硬件问题和系统奔溃
    - 多事务的**并行**执行
  - 事务开始和结束的时候数据库都必须是consistent的
  - 事物的四个性质ACID：
    - 事务的原子性 **Atomicity**
      - 事务中的所有步骤只能完全执行(commit)或者回滚(rollback)
    - 事务的一致性 **Consistency**
      - 单独执行事务可以保持数据库的一致性
    - 事务的独立性 **Isolation**
      - 事务在**并行执行**的时候不能感知到其他事务正在执行，执行中间结果对于其他并发执行的事务是隐藏的
    - 事务的持久性 **Durability**
      - 更新之后哪怕软硬件出了问题，更新的数据也必须存在
  - A simple Transaction Model
    - read(X)
    - write(X)
- 事务的状态



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

- committed 事务被完整的执行

5.1.2 事务的并发执行

- 并发执行的异常
  - Lost Update丢失修改
    - 不能同时改

T1	T2
Read A (100)	
	Read A (100)
A = A-1 (99)	
	A=A-1 (99)
Write A (99)	
	Write A (99)

  - Dirty Read脏读
    - 修改了但还没有提交的数据

T1	T2
Read A (100)	
A = A-1 (99)	
Write A (99)	
	Read A (99)
	A=A-1 (98)
rollback	
	Write A (98)
	commit

  - Unrepeatable Read不可重复读

T1	T2
Read A (100)	
	Read A (100)
	A=A-1 (99)
	Write A (99)
Read A (99)	

    在一个进程中，没有修改动，读两次数值不一样
  - Phantom Problem幽灵问题

T1	T2
select * from student where age=18 (100 records with age=18)	
	insert into student(id, gender, age) values('008','M', 18) (Add a new record with age=18)
select * from student where age=18 (101 records with age=18 !!!)	

- 同时执行多个事务，可以**提高运行的效率**，减少平均执行时间
  - 并发控制处理机制：让并发的事务独立进行，控制并发事务之间的交流
  - Schedules 调度
    - 一系列用于指定并发事务的执行顺序的指令
      - 需要包含**事务**中的所有指令
      - 需要保证单个事务中的指令的相对顺序
    - 事务的最后一步
      - 成功执行，最后一步是commit instruction
      - 执行失败最后一步是abort instruction
    - serial schedule **串行调度**：一个事务调度完成之后再进行下一个
    - equivalent schedule 等价调度：改变处理的顺序但是和原来等价
    - **Serializability**可串行化
      - 基本假设：事务不会破坏数据库的一致性，只考虑读写两种操作
      - 冲突可串行化 conflict serializability
        - 同时读不引发冲突，而读写并行或者同时写会引发冲突
        - **conflict equivalent**：两个调度之间可以通过**改变一些不冲突的指令来转换**，就叫做冲突等价
        - **conflict serializable**：冲突可串行化：当且仅当一个调度S可以和一个串行调度等价
        - **Precedence graph** 前驱图
          - 图中的顶点是各个事务，当事务T_i, T_j 冲突并且T_i 先访问出现冲突的数据的时候，就画一条边$T_i \rightarrow T_j$
          - 一个调度是冲突可串行化的当且仅当前驱图是无环图
          - 对于无环图，可以使用**拓扑逻辑排序**获得一个合适的执行顺序
    - Recoverable Schedules 可恢复调度
      - database must ensure that schedules are recoverable. 不然会出现dirty read

        - 如果一个事务T_1要读取某一部分数据，而T_2要写入同一部分的数据，则T_1必须在T_2commit之前就commit，否则就会造成dirty read
    - **Cascading Rollbacks** 级联回滚
        - 单个事务的fail造成了一系列的事务回滚
    - **Cascadeless Schedules** 避免级联回滚的调度
        - 对于每一组事务a和b并且b需要读入一个a写入的数据，那么a必须在b的读操作开始之前commit
            - 不能读脏数据
        - **Cascadeless Schedules**也是可恢复的调度
- 事物隔离级别
    - 快照隔离snapshot isolation

5.1.3 Concurrency Control 并发控制

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

	S	**X**
S	true	flase
X	false	false

        - 如果请求的锁和其他事务对这个数据项已经有的锁不冲突，那么就可以给一个事务批准一个锁
        - 对于一个数据项，可以有任意多的事务持有S锁，但是如果有一个事务持有X锁，其他的事务都不可以持有这个数据项的锁
        - 如果一个锁没有被批准，就会产生一个请求事务，等到所有冲突的锁被release之后再申请
- 锁协议中的特殊情况
    - dead lock 死锁：两个事务中的锁互相等待造成事务无法执行，比如事务2的锁需要事务1先release，但是事务1的release步骤在事务2的申请锁后面，就会造成事务12的死锁

    - Starvation 饥荒：一个事务在等一个数据项的Xlock，一群别的事务在等他release，造成饥荒
- Two-Parse Locking Protocol 二阶段锁协议：
    - 两个阶段 growing和shrinking，growing只接受锁而不释放，shrinking反之
    - 可以证明两阶段封锁协议确保冲突可串行化的调度（充分但不必要）；对于任何事务，在调度中该事务获得其最后加锁的位置（增长阶段结束点）称为事务的封锁点（lock point），多个事务可以根据他们的封锁点进行排序
    - 无法解决死锁的问题，可能出现级联回滚
    - **strict two-phase locking**
        - 每个事务都要保持所有的exclusive锁直到结束；防止其他事务读脏数据
        - 为了解决**级联回滚**的问题
    - **Rigorous two-phase locking**
        - **所有的锁**必须保持到事务commit或者abort
- Lock Conversions锁转换：提供了一种将S锁升级为X锁的机制
    - 两个阶段
        - 第一个阶段可以设置S和X锁，也可以升级S锁
        - 第二个阶段可以释放S和X锁，也可以降级X锁
    - 事务不需要显式调用所得请求，自动会为事务产生加锁、解锁指令，比如read和write的执行过程如下
        - 所有的锁在事务commit或者abort之后再被释放
        - 在read后，系统就产生一条lock-S指令，read随后
        - 在write后，检查是否有共享锁，有，就先更新，再write；无，就先发出lock-X，再write

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

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

- 锁的实现：Lock Manager可以被作为一个独立的进程统一来接收事务发出的锁和解锁请求
  - Lock Manager会回复申请锁的请求
  - 发出请求的事务会等待请求被回复再继续处理
  - lock manager维护一个内存中的数据结构lock-table来记录已经发出的批准
    - Lock table 是一个**in-memory的hash表**
    - 通过被上锁的数据项作为索引，黑框代表上锁，而白框表示在等待
    - 新的上锁请求被放在队列的末端，并且在和其他锁兼容的时候会被授权上锁
    - 解锁的请求会删除对应的请求，检查后面的请求是否可以被授权
    - 如果一个事务aborts了，所有该事务的请求都会被删除
    - lock-manager会维护一个记录每个事务上锁情况的表来提高操作的效率
- Deadlock prevention protocols 死锁保护协议，保证系统不会进入死锁
  - predeclaration 执行之前先检查会不会出现死锁，保证一个事务开始执行之前对涉及到的所有的数据项都上锁
  - graph-based protocol：使用**偏序**来确定数据项上锁的顺序
  - **wait-die** scheme 被动
    - 基于非抢占技术
    - 老的事务等待新事务释放，但是新的事务不等老的而是直接回滚
  - **wound-wait** scheme 主动
    - 基于抢占技术
    - 老的事务强制让新的事务回滚而不等待其释放，新的事务会等老的事务结束
  - Timeout-Based Schemes

        - 基于锁超时
    - 只等待一段时间，过了时间就回滚
    - 容易实现，但是会导致starvation
    - Deadlock Detection 死锁检测
        - wait-for 图: 所有的事务表示图中的点，如果事务i需要j释放一个data item则图中画一条点i到点j的有向边，如果图中有环，说明系统存在一个死锁——跟前驱图很相似
        - 死锁恢复
            - total rollback 将事务abort之后重启
                - partial rollback 不直接abort而实仅回滚到能解除死锁的状态
            - 同一个事务经常发生死锁会导致starvation，因此避免starvation的过程中cost要考虑回滚的次数
- **Multiple Granularity** 多粒度

    > 把多个数据聚为一组，将他们作为一个同步单元

    - 允许数据项具有不同的大小，并定义**数据粒度的层次结构**，其中小粒度嵌套在大粒度中
    - 可以用树形结构来表示
    - 锁的粒度 (level in tree where locking is done)
        - 细fine granularity(lower in tree) 高并发，高开销
        - 粗coarse granularity(higher in tree) 低并发，低开销
        - 最高等级的是整个DB
- 最低等级的是区域，文件和记录
- 当事务对一个结点加锁，或为共享锁，或为排它锁，该事务也以同样类型的锁隐式地封锁这个结点的全部后代结点。
- 扩展的Lock Modes

    > 做标记，我在下面有了锁了
    >
    > 如果是shared或exclusive锁，则不会再表示下面的信息，因为就表示后代节点都锁了

    - **intention-shared** (IS): indicates explicit locking at a lower level of the tree but only with shared locks.在树的较低层进行显式加共享锁
    - **intention**-exclusive (IX): indicates explicit locking at a lower level with exclusive or shared locks.在树的较低层进行显式加排他锁
    - **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.在该节点显式的加共享锁，更低层显式的加排他锁
    - 冲突矩阵如下

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

> 多粒度封锁协议
>
> - 加锁自顶向下
> - 锁的释放自底向上

多版本并发机制

- 每一次写都不是覆盖，而是生成一个新版本
- 保证用于读取的版本的选择，就可以保证可串行性
- 版本号会改变

多版本两阶段封锁协议

> 将多版本并发控制和两阶段封锁结合

- 对只读事务和更新事务加以区分
- 执行强两阶段封锁协议

5.2 Recovery System 事务恢复

- 故障的分类
    - Transaction failure 事务错误
        - 逻辑错误
        - 系统错误，死锁属于系统错误
    - System crash 系统崩溃导致的故障(磁盘没出事)
        - 非易失性存储器没事，易失性会丢失
    - Disk failure 磁盘中的问题导致磁盘存储被销毁

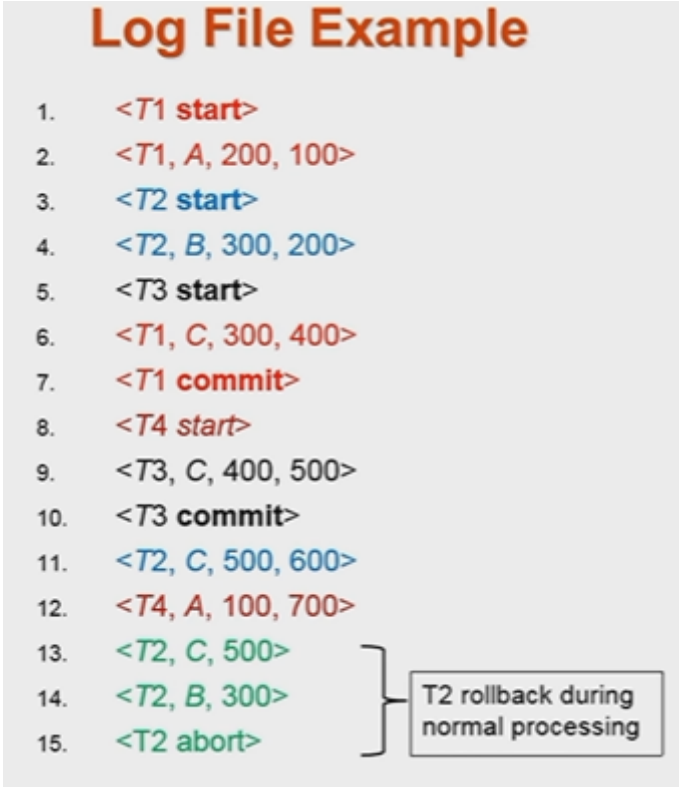
> 大多是基于日志的恢复

- 恢复算法：保持数据库的一致性，事务的原子性和持久性。由两部分组成：

    - 在普通事务处理中要保证有足够的信息保证可以从故障中恢复
    - 在故障发生之后要保持数据库的一致性，事务的原子性和持久性
- 存储器
    - 易失性存储器
    - 非易失性存储器
    - 稳定存储器：理想，维护**多个拷贝存储器**
- Data Access 数据访问回顾
    - 物理block是磁盘上的区分
    - 缓冲block是在主存中的block
    - 磁盘和主存之间的数据移动依赖input和output操作
    - 每个事务T_i在内存中有自己的work-area，并且**拷贝**了一份该事务要用到的全部数据
    - 事务通过read和write操作把数据在自己的工作区域和buffer blocks区间之间进行传递
- 如何在事务failure的情况下仍然保证原子性
    - 先把数据存储在磁盘上，而不是直接存到数据库中
    - 然后在提交点对数据库进行修改，如果发生错误就立马回滚
        - 但这个方法效率太低了，是世上没有被采用

5.2.1 log-based Recovery基于日志的恢复

- 日志(log)被存储在稳定的存储中，包含一系列的日志记录（流水账）
    - 事务开始`<T start>`
    - 写操作之前之前的日志记录`<Ti,X,V1,V2>`(X)是写的位置，V1，V2分别是写之前和之后的X处的值
    - 事务结束的时候写入`<Ti commit>`
    - 事务中止写入`<Ti abort>`

    - 

- 更新事务导致的不一致性
    - 新的数据在提交的时候不一定是安全的：错误发生时难以保护改变后的值不变
    - 旧的数据在提交之前不一定是安全的：在commit之前发生错误将无法回滚到原来的值
    - 对于更新事务的两条规则
        - commit rule：新的数据在commit之前必须被写在**非易失性**的存储器中
        - logging rule：旧的值在新的写入之前需要被写在日志里
    - 日志中写入commit的时候视为一个事务被提交了，但此时**buffer中可能还在进行write**操作，因为**log的写入先于操作**，所以不一定修改好了。checkpoint是直接写到disk里了
- **deferred database modification** 延迟数据库更新：先把所有的更新写在日志里，在写入commit之后再开始写入数据库
    - 假设事务是串行执行的
    - 事务开始的时候要写入`<Ti start>`
    - **write**(*X*) 操作对应的日志是`<Ti, X, V>`，V表示X新的值
    - 事务partially commits的时候需要写入commit
    - 然后根据日志来实际执行一些write的操作
        - 当错误发生时，当且仅当日志中start和commit都有的时候，事务需要redo
- **immediate database modification** 直接修改数据库

    - 先要写好日志记录，假设日志记录直接output到稳定的存储中
    - block的输出可以发生在任何时间点，包括事务commit前和commit后，block输出的顺序和write的顺序不一定相同
    - 恢复的过程中有两种操作
        - undo：撤回，将事务T_i 中已经更新的值变回原本的值；改成旧值
        - redo：从事务T_i 的第一步开始重新做，将所有值设定为新的值；改成新值
        - 两种操作都需要**idempotent**——也就是操作执行多次和执行一次的效果相同
    - undo的条件：日志中包含这个事务的start而不包含commit，即事务进行到一半中断了
    - redo的条件：日志中包含这个事务的start和commit
- 并行控制和恢复
    - 所有事务共用一个日志和disk buffer
    - 基本的假设
        - 如果一个事务改变了某个数据项，其他的事务直到这个事务commit或者abort之前都不能改变这个数据项的值
        - 没有commited的事务引起的更新不能被其他事务更新
    - 日志中不同事务的日志可能会相交
- check point
    - 通过定期执行checkpoint操作来完成**简化**的恢复

      > 1.将内存中的记录都写到稳定的存储中
      >
      > 2.将所有更改过的block写入磁盘中
      >
      > 3.写入日志记录< **checkpoint L**>，其中L是一个在checkpoint时依然处于进行状态的事务

    - 通过checkpoint，在日志处理的时候就不需要处理所有的日志，只需要**关注异常时正在活跃的事务**，恢复的步骤如下
        - 从日志的末尾向前扫描，直到发现最近的checkpoint记录
        - **只有L中记录的，或者在L之后发生的事务需要redo或者undo**
        - checkpoint之前的记录已经生效并保存在了稳定的存储中
    - 日志中更早的部分**或许需要**undo，但一定不需要redo
        - 继续向前扫描直到发现一个事务的start日志
        - 最早的start之前的日志不需要进行恢复操作，并且可以清除
- 恢复算法

    - 单个事务回滚时的基本操作
        - 从后往前扫描，当发现记录$< T_i, X_i, V_1, V_2 >$的时候
        - 将X的值修改为原本的值
        - 在日志的末尾写入记录$< T_i, X_i, V_1 >$
        - 发现start记录的时候，停止扫描并在日志中写入abort记录
    - 恢复的两个阶段：redo和undo
    - redo需要先找到最后一个check point并且设置undo-list

      > 1.从**checkpoint**开始往下读
      >
      > 2.当发现修改值的记录的时候，redo一次将X设置为新的值
      >
      > 3.当发现start的时候将这个事务加入undo-list
      >
      > 4.当发现commit或者abort的时候将对应的事务从undo-list中移除

    - undo

      > 1.从日志的**末尾**开始往回读
      >
      > 2.当发现记录 <Ti, Xj，V1，V2>并且Ti在undo-list中的时候，进行一次回滚
      >
      > 3.当发现Ti start并且Ti在undo-list中的时候，写入abort日志并且从undo-list中移除Ti
      >
      > 4.当undo-list空了的时候停止undo

- log record buffering 缓冲日志记录
    - 日志记录一开始在主存的缓冲区中，当日志在block中满了的时候或者进行了log force操作(强制的写，将缓冲的日志写到磁盘，上面提到的checkpoint)时写入稳定的存储中
    - 增加吞吐率，掉电会丢失
    - 需要遵守的规则
        - 写入稳定存储中的时候日志记录按照原本的顺序写入
            - 在commit记录被写入稳定存储的时候，Ti才算进入commit状态
        - WAL(write-ahead logging)规则：在数据block写入数据库之前，必须先把日志写入稳定的存储中
- Database buffer
    - force —— no-force
        - 事务在提交时强制地将修改过的所有的块都输出到磁盘
        - 即使一个事务修改了某些还没有写回磁盘的块，也允许提交
    - steal —— no-steal

        - 允许将修改过的块写到磁盘，即使做这些修改的事务没有全部提交
        - 一个仍然活跃的事务修改过的块都不应该写出到磁盘
    - fuzzy checkpoint 模糊检查点
        - 允许在checkpoint记录写入日志后，在修改过的缓冲块写到磁盘前开始做更新
- 非易失性存储器数据丢失的故障
    - 把整个数据库的内容转储

5.2.2 ARIES Recovery Algorithm——Aries恢复算法

> 工业级的

- 和普通恢复算法的区别：
    - 最核心的区别——**Aries算法考试考到的概率很高**
    - 使用LSN(log sequence number)来标注日志
        - 日志编号
        - 以页的形式来存储LSN来标注数据库页表中进行了哪些更新
    - 支持物理逻辑redo
    - 使用脏页表(dirty page table)
        - 最大限度的减少恢复时不必要的重做
    - 模糊的checkpoint机制，只记录脏页信息和相关的信息，不需要把脏页写入磁盘
- ARIES中的数据结构
    - Log sequence number (LSN)
        - 用于标识**每一条记录**，需要是线性增长的
        - 其实是一个offset，方便从文件的起点开始访问
    - **Page LSN** 每一页的LSN
        - 是每一页中**最后一条 起作用**的日志记录的LSN编号
        - 每当一个更新的操作在某一页发生的时候，Page LSN就变成对应的Page LSN
        - 在恢复的撤销阶段，LSN值不超过PageLSN的日志记录将**不会在该页上执行**，因为其动作已经在该页上了
        - 可以避免重复的redo
    - log record 日志记录
        - 每一条日志记录包含自己的LSN和**同一个事务中前一步操作**的LSN——PrevLSN
        - CLR：在恢复期间不需要undo，是redo-only的日志记录

        - 补偿日志记录
        - 有一个UndoNextLSN区域用于记录下一个(更早,往前搜索)的需要undo的记录
        - 在这之间的记录应该早就已经undo了
    - Dirty Page Table 脏页表
        - 存储在缓冲区的，记录已经被更新过的page的表
        - 包含以下内容
            - 每个页的PageLSN
            - 每个页的RecLSN，表示这一页的日志记录中，LSN在**这**之前的记录已经**都被写入磁盘中**了
                - 当page被插入脏页表的时候，**初始化为当前的PageLSN，RecLSN的值被设置成日志的当前末尾**
                - 记录在checkpoint中，用于减少redo的次数
        - 只要页被写入磁盘，就从脏页表中移除该页
    - checkpoint处的日志记录
        - 包含：脏页表和当前活跃的所有事务
        - 对每一个活跃的事务，记录了**LastLSN**，即这个事务在日志中写下的**最后一条记录**
        - 在checkpoint的时间点，脏页的信息不会写入磁盘
- ARIES算法的恢复操作
    - 分为三个阶段：**分析阶段，redo阶段和undo阶段**
        - RedoLSN 记录了从哪一条开始需要redo
    - 分析阶段：需要决定哪些事务undo，哪些页是脏页以及redo从哪里开始
        - 从**最后一条完整的checkpoint日志记录**开始
        - 读取脏页表的信息
            - 设置RedoLSN = min RecLSN(脏页表中的)，如果脏页表是空的就设置为checkpoint的LSN
            - 设置undo-list：checkpoint中记录的事务
            - 读取undo-list中每一个事务的最后一条记录的LSN
        - 从checkpoint开始正向扫描
            - 如果发现了不在undo-list中的记录就写入undo-list
            - 当发现一条**更新记录**的时候，如果这一页**不在脏页表**中，用该记录的LSN作为**RecLSN**写入脏页表中
            - 如果发现了标志事务**结束**的日志记录(commit, abort) 就从undo-list中**移除**这个事务
            - 搜索直到undo-list中的每一个事务都到了最后一条

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