DBS Review

Lecture 1: Introduction

无用跳过

Lecture 2： Relational Model

Structure

- relation -> table
  - 无序(irrelevant)、不重复(No duplicated)、原子性(atomic)
- turple -> row
- attribute -> column name 列名
- attribute value -> domain
  - atomic:不可分割

Relation Schema 关系模式： $R = (A_1, A_2, \dots A_n), A_i$一系列属性

- $r(R)$代表一种关系（一张表）

Relation Instance关系实例：relation的快照，具体的值

Key 键

Superkey 超码：元组的唯一标识，不能重复，可以存在冗余

Candidate Key 候选码：不含冗余属性的超码

Primary Key主码：用户指定的候选码

Foreign Key外码：A中包含B的主码b，于是b是A的外码**（注意英文表述）**：

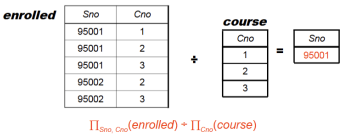
- a foreign key from r1 referencing r2
- r1 is a referencing relation
- r2 is a referenced relation

- **箭头 外键指向主键**

Schema Diagram：少见，就是一个外键指向主键的图

Relational Algebra Operations

六基本操作	表达式	解释
Select 选择	$\sigma_p(r)$	p:筛选的条件，r是对应的表返回满足条件的行
Project 投影	$\Pi_{A_1,..A_n}(r)$	投影出只含有A这些属性的表，**删除重复的行**
Union 并	$r \cup s$	将两个属性数相等且所有属性的域相同的两个表合并，**并去重**
Set defference 差	$r - s$	返回属于关系 却不出现在关系中的元组的关系
Cartesian Product 笛卡尔积	$r \times s$	两张表做笛卡尔积，**去除相同属性或者重命名属性**
Rename 重命名	$\rho_x(E)$ $\rho_{x(A_1,A_2....A_n)}(E)$	重命名，第二种顺带修改属性名称

拓展操作	表达式	解释
Set intersection	$r \cap s$	取出公共元素$r-(r-s)$
Natural join	$r \bowtie s$	保证两边公共属性相等的连接，**删除同名属性**
Theta join	$r \bowtie_\theta s$	满足条件的自然连接 $\sigma_\theta(r \times s)$
Division	$r \div s$	取出r中除了s中属性之外的属性固定，完整取完一个s的属性  解决"**for all** 的问题"
Assignment	$temp \leftarrow r \times s$	临时变量赋值操作
Generalized Projection	$\Pi_{F_1..F_n}(E)$	F可以表示数值的四则运算

拓展操作	表达式	解释
Aggregate 聚集	$G_1,G_2...G_n g_{F_1(A_1),F_2(A_2)...F_n(A_n)}(E)$	G的值代表分组(按照这个属性分组之后再计算函数)，F的值是代入函数的值 sum/max/min/avg/count/...

运算符优先级： project > select > cartesian product > join division > intersection > union difference

Deletion:$r \leftarrow r - E$

Insertion:$r \leftarrow r \cup E$

Update:$r \leftarrow \Pi_{F_{1..}}(r)$

Lecture 3 :SQL

数据类型： char(n) varchar(n) int smallint numeric(p,d)(p位数字(加1位符号位)，d位在小数点右边） real :float double float(n) date time timestamp 、blob(20MB)二进制文件 、clob（10kb）字符文件

create table:

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

完整性约束

- primary key(A1,A2...An):指定属性为主码，是非空的
- foreign key(A1.A2...An)references S声明外码，声明A的属性取值，必须和S中对应属性取值保持一致
- not null非空
- check(P):对于数据进行约束

drop and alter table index： 不常考指令

```
DROP TABLE r;  //从数据库中完全删除该表
DELETE FROM r; //只是删除所有元组，保留属性
ALTER TABLE r ADD A D;  //在表中添加属性
ALTER TABLE r ADD (A1 D1,A2 D2..);
ALTER TABLE r DROP A; //删除属性A
CREATE INDEX <i-name> ON <table-name> (<attribute-list>); //建立一个索引
Eg.
create index b_index on branch (branch_name);
CREATE UNIQUE INDEX <i-name> ON <table-name> (<attribute-list>);//将该索引声明为候选键
DROP INDEX <i-name>  //删除索引
```

查询语句

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

- from 对于表的层面 选择所需要的表
- where 对于表中的数据进行一个筛选
- group by 分组 使用聚集函数 不分组默认全部
- having 现在的表是所有分组都有的 可以分组进行筛选 或者 对于聚集函数计算出的值进行筛选 最后的筛选
- select 选择出其中需要展示的属性

```
SELECT A1 A2... //选择属性
FROM R1 R2 R3    //选择表
WHERE P             //限制条件

SELECT distinct  //去除重复
SELECT all           //不去除重复 默认all

SELECT *
SELECT A*10 //允许计算

//重命名
old_name as new_name  //as有时可以省略

//字符串操作(不是重点)
WHERE A LIKE '%a'  //使用like 任何子串
WHERE A LIKE '_a'  //使用like 任何字符
WHERE a LIKE '\%ABC' escape '\'; //把\当作转意字符 来显示%
```

```
SELECT 'a='||name    //表达字符串连接 结果为 'a=jdshf'
upper() //把小写改写成大写
lower() //把大写改写为小写

```

- select:默认不去重
- from：表的笛卡尔积
- where：中使用 and or not和between...and..

只有SELECT count()默认不去除重复

```
//次序
order by A1,A2    //多个属性时,先按照第一个排列 如果有重复再按照第二个排列
order by A1 desc //按照降序排列
order by A2 asc //按照升序排列
```

- 集合操作：连接两个select的表

```
union                 //交操作 默认去重
union all           //不去重
intersect           //并
intersect all
except                //差
except all
```

- 聚集函数⭐

```
avg(col): average value
min(col): minimum value
max(col): maximum value
sum(col): sum of values
count(col): number of values // count(*)忽略空值
count (distinct col)
```

select 中出现的属性一定是在group by中出现的

```
SELECT <[DISTINCT] c1, c2,...>
FROM <r1, ...>
[WHERE <condition>]
[GROUP BY <c1, c2, ...> [HAVING <cond2>]]
[ORDER BY <c1[DESC] [, c2[DESC|ASC], ...]>]
```

可以使用 distinct 关键字来去重（count(*) 时不行）

例子：

```
Find the names of all branches located in city Brooklyn where the average account balance is more than $1,200.
select A.branch_name,avg(balance)
from account A,branch B
where a.branch_name=b.branch_name and branch_city="Brooklyn"
group by A.branch_name
Having avg(balance)>1200
```

空值判断

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

`is unknown / is null` 可以用来测试是否为未知/空值

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

嵌套查询

```
WHERE name in/not in (SELECT ......) //在其中或者不在其中
WHERE A > some (SELECT .......) //比之间的一个大或者小就行
WHERE a > all (SELECT ........) //比其中所有的大或者小
exists //存在返回1
not exists //不存在返回0
unique  //只出现一次 返回1 多次出现返回0
not unique //出现多次 返回 1 出现一次返回0
```

视图

```
CREATE VIEW <v_name> (c1, c2, ...) AS
SELECT e1, e2, ... FROM ...
DROP VIEW <V_NAME>

select ..
from (select....)as ....//必须给出别名
where...

with ...(value) as(select ....)
select ... from ...where...
```

- **with语句 建立一个临时视图**

删除插入更新语句

```
DELETE FROM <TABLE>
WHERE [CONDITION]

UPDATE <table | view>
SET <c1 = e1 [, c2 = e2, ...]>
[WHERE <condition>]

UPDATE account
SET balance = case
   when pred1 then result1
   when pred2 then result2
  ...
   else resultn
   end

#单一插入
INSERT INTO <table|view> [(c1, c2,...)]        //表的属性可以省略
VALUES (e1, e2, ...)     //插入时没写的数据自动赋值为null
# 多元素插入
INSERT INTO <table|view> [(c1, c2,...)]
SELECT e1, e2, ...        //只执行一次
FROM ...
```

不支持多个表自然连接 再删除。删除多个表信息时，应当一个一个删除。

- 在同一SQL语句内，除非外层查询的元组变量引入内层查询，否则层查询只进行一次。
- 先计算平均值
- 之后再与平均值比较，不会重复计算平均值

  DELETE FROM account
  WHERE balance < (SELECT avg(balance)FROM account)

其他

行列视图：建立在基本表上的视图，视图的列对应表的列，**只有行列式图能更新**

事务Transactions

```
begin atomic
....
end
commit
rollback
```

连接关系

> *Inner Join 保留两边都有*
>
> *left join 保留左边全部*
>
> *right join 相反*
>
> *full join 保留全部*
>
> *（inner/outer）join optional*
>
> *outer：不能匹配补充null*

自然连接：Rnatural {inner join, left join, right join, full join} S（**去除重复属性**）

非自然连接：R {inner join, left join, right join, full join} S+ on<条件判别式>（**不去除重复属性**）/ +using <属性名>（**去除重复属性**）

Lecture 4: Advanced SQL

定义域和类型

```
Create type person_name as varchar (20)
Create domain Dollars as numeric(12, 2) not null;
Create domain Pounds as numeric(12,2) constraint value-test
check(value>100) ; //定义域检测
```

> *域可以添加声明约束：not null等*
>
> *域的类型可以被用于其他域的定义，类型只能用基础的定义*

Integrity Constraints 完整性控制

Referential Integrity引用完整性：

A，B两个关系，B是引用关系，A是被引用（A主键，B外键）

- **Insert**：插B，检查A

- **Delete**：删A，检查B
- **Update**：改A，检查B，改B，检查A

```
foreign key (branch_name) references branch
//同时删除 A删除 B同时
on delete casecade
//同时更新 A更新 B也更新
on update casecade
//设置空 或默认
on delete set null/default
on update set null/default
```

Assertions

```
CREATE ASSERTION<assertion-name>
CHECK <predicate>;   # 永远保证为真
```

> *对于任意 x p(x) 为真*
>
> *通常使用不存在 一个x 使得p(x)为假(not exist)*
>
> *not exists X such that not P(x)* ***使用not exist***

Triggers 触发器

```
Create trigger overdraft-trigger after<before>
<对于每一行操作 eg. update of <table_name> on
<arrtibute_name>/delete/insert on   <table_name>>
Referencing old row as ... //for deletes and updates
Referencing new row as ... //for inserts and updates
inserted/deleted # 代替nrow和orow
for each row
when<条件>
begin
...
end;
```

Authorization 授权控制

数据库权限种类：read insert update delete

数据库模式权限：index resource alteration drop

```
GRANT <privilege list> ON <table | view>
TO <user list>
grant select on branch to U1with grant option; //同时获得分发权限的权限
```

支持创建角色

```
Create role teller;
Grant select on branch to teller;
//支持角色分发
Grant teller to manager;
Grant teller to alice, bob;
```

回收权限

```
REVOKE<privilege list> ON <table | view> FROM <user list> [restrict（只收回自己的） | cascade(连级收回)]
```

limitations

- SQL does not support authorizationat a tuple level.

审计语句：查看记录

```
AUDIT <st-opt> [BY <users>] [BY SESSION | ACCESS] [WHENEVER SUCCESSFUL | WHENEVER NOT SUCCESSFUL]
audit table by scott by access whenever successful
```

嵌入式 动态 ODBC JDBC 应该不考 不看了

Lecture 5: Entity-Relationship Model

Entity Sets and Relationship Sets 实体集、关系集

Entity set : “类(class)”。实体由一系列 **属性** 描述，而实体集是由一系列相同性质 或属性的实体组成的集合。

Keys：和之前一致

Attribute：

- **simple composite 简单和复合属性**
- **single-value and multi-valued 单值属性 多值属性**
- **derived attribute 派生属性** 能通过其他属性计算得出，如根据出生时间推算年龄，不需要单独存储

RelationShip Set：实体集之间的联系

- **Degree** 连接的实体集的数量

- 二元关系：一对一、 一对多、多对一、多对多**（判断可以通过画图简单）**
- **Key**：关系集的**超码**是连接的实体集的**超码**的组合

E-R Diagrams

角色role：在关系实例的连线上可以表达角色

Cardinality Constraints映射基数

- ← 表示1 ， – 表示多

> *三元关系中： 箭头只能出现一次， 否则会出现 二义性*

- **Total participation 全参与**= :实体集全部参与到关系中
  - a. . b表示 一个对象能对应另一边对象的范围 *表示未知上界
  - **下限用于区分全参与或者是部分参与**

Weak Entity Set 弱实例集

没有主码的集合，**依附于强实体集存在**，存在一个分辨符(discriminator) 下划虚线 *主码=强实体集合主码+分辨符*

Extended E-R features

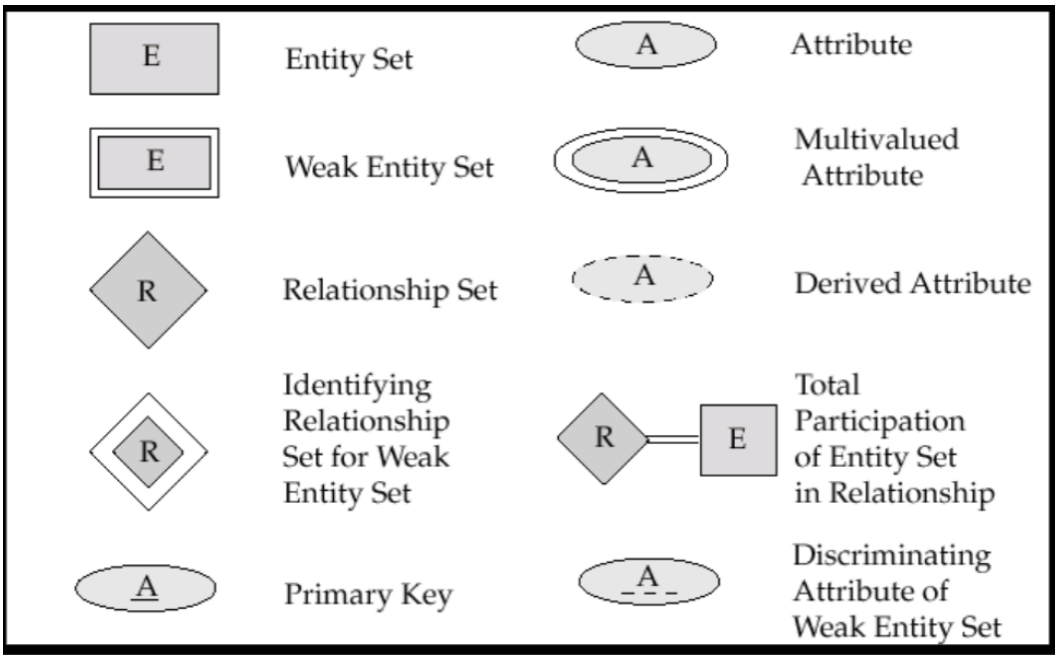
Specialization (特殊化、具体化)： 自顶向下的设计过程，画图的方式就是从上往下画，Entity的内容逐渐细分，但是都继承了上一阶的所有**attribute** 类似于**oop继承**

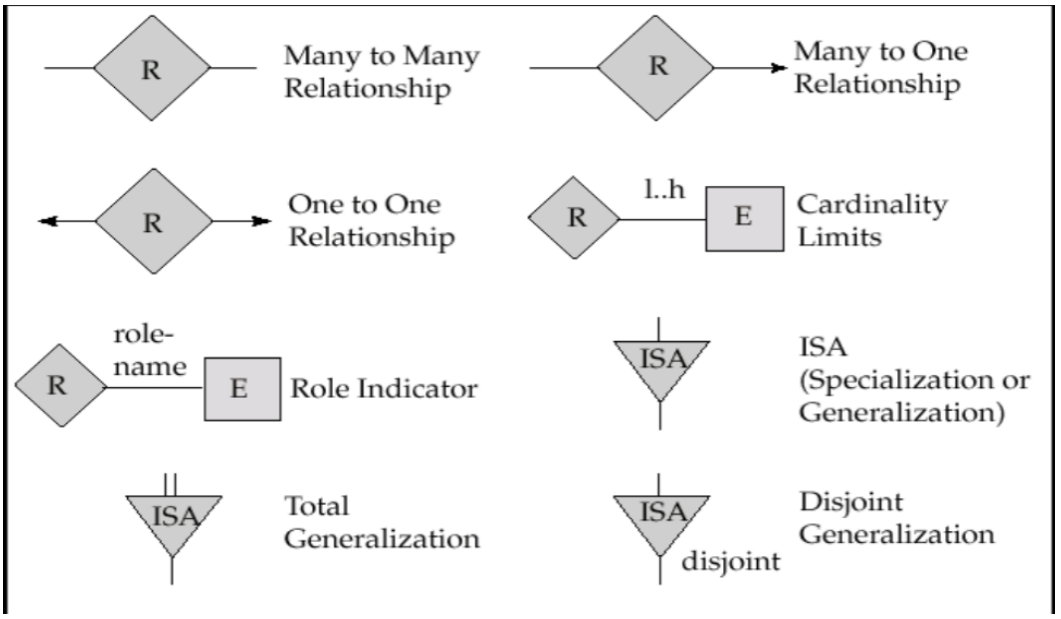
Generalization (泛化、普遍化) 从下往上，下层的内容合成上层的内容

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

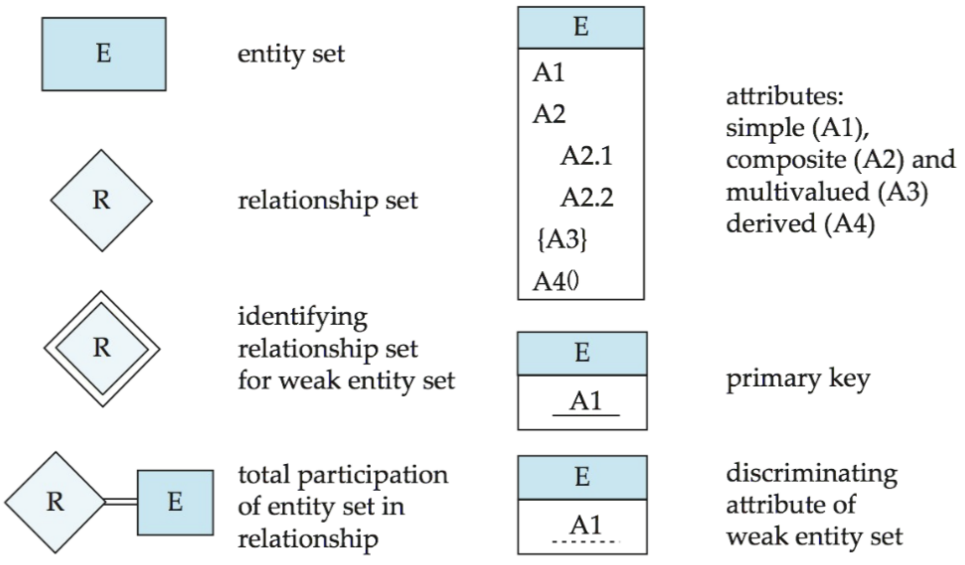
两种表达方式

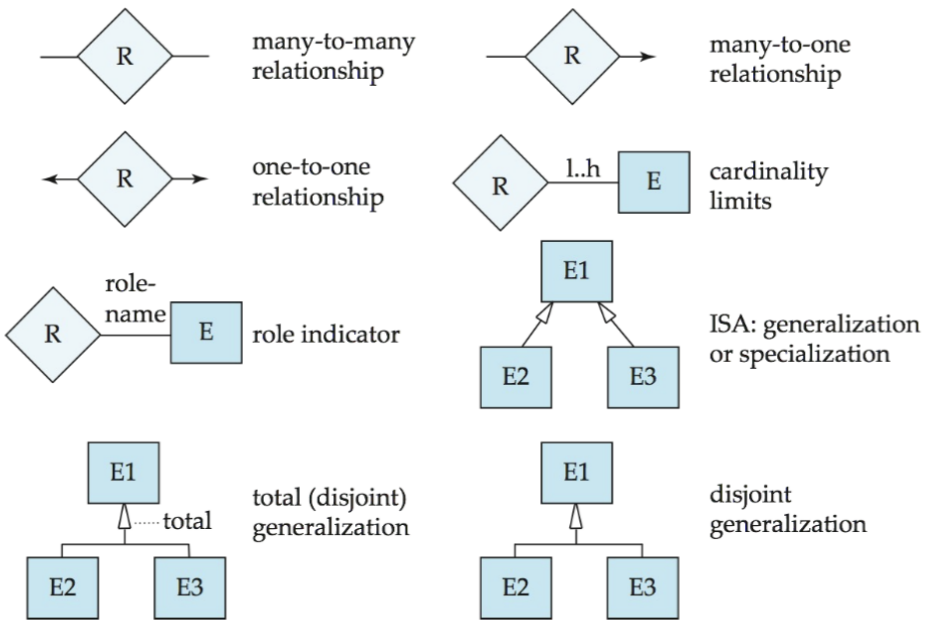
第一种：


E
Entity Set
A
Attribute
E
Weak Entity Set
A
Multivalued Attribute
R
Relationship Set
A
Derived Attribute
R
Identifying Relationship Set for Weak Entity Set
R
E
Total Participation of Entity Set in Relationship
A
Primary Key
A
Discriminating Attribute of Weak Entity Set


R
Many to Many Relationship
R
Many to One Relationship
R
One to One Relationship
R
l..h
E
Cardinality Limits
role-name
R
E
Role Indicator
ISA
ISA (Specialization or Generalization)
ISA
Total Generalization
ISA
disjoint
Disjoint Generalization

第二种:


E
entity set
E
A1
A2
A2.1
A2.2
{A3}
A4()
attributes:
simple (A1),
composite (A2) and
multivalued (A3)
derived (A4)
R
relationship set
R
identifying relationship set for weak entity set
E
A1
primary key
R
E
total participation of entity set in relationship
E
A1
discriminating attribute of weak entity set



E-R Schema to Table

- **强实例集**直接转化为表
  - **composite attributes**复合属性直接展开
  - **multivalued attribute**：多值属性 新建一个表格，用原表**主键**加上**多值属性**构成
- **弱实例集**用主键加上弱连接 不用表示弱连接关系
- **关系集合**：用两张表的主键作为属性建表
  - **多的那边作为主键**
  - **对于一对多多对一，可以将关系合并到多的实体表中**
- **特化关系**：冗余记录 或者 主键加特殊属性

Lecture 6: Relational Database Design

Norm Form

First Normal Form: 只要所有属性的域是原子性的就是第一范式，在关系数据库中，必须符合第一范式。

- 缺点：冗余、更新复杂

Decomposition 分解:

- ⭐ Lossless-join decomposition (无损连接分解):

- $R = R_1 \cup R_2, r = \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r)$
- **自然连接之后是原表，只要不等于都是有损分解**
- 🌟**判定**：公共属性是某一张表超码：$R_1 \cap R_2$ 是R_1/R_2超码

Functional Dependencies 函数依赖

- 在R上有函数依赖$\alpha \to \beta$ 当且仅当 r上有两个元组$t_1 t_2$,如果$t_1[\alpha] = t_2[\alpha]$则$t_1[\beta] = t_2[\beta]$
- 如果K是关系r的主键，则一定有$K \to R$
- **Trivial dependency 平凡依赖**：$\beta \subseteq \alpha \to (\alpha \to \beta)$

Closure 闭包

set of functional dependence

原始的函数依赖能够推出的所有函数依赖的集合 就是F闭包（F^*）

> *reflexivity*： $\beta \subseteq \alpha$ *then* $\alpha \to \beta$
>
> *augmentation* :$\alpha \to \beta$ *then* $k\alpha \to \beta k$ $\alpha k \to \beta$
>
> *transitivity*:$\alpha \to \beta$ *and* $\beta \to k$ *then* $\alpha \to k$
>
> *union*:$\alpha \to \beta$ *and* $\alpha \to k$ *then* $\alpha \to \beta k$
>
> *decomposition*:$\alpha \to \beta k$ *then* $\alpha \to \beta$ *and* $\alpha \to k$
>
> *pseudotransitivity* :$\alpha \to \beta$ *and* $k\beta \to \theta$ *then* $\alpha k \to \theta$

set of attribute set

在F下由a所直接和间接函数决定的属性的集合称为a^+

如果闭包包含所有属性，则a是候选键

Canonical Cover 正则覆盖

- 不存在多余的函数依赖
- **extraneous attributes无关属性**：去除某一个属性之后不改变闭包

判断候选码 和找正则覆盖都通过画图完成

🌟**BCFN**

定义：闭包中任何一个函数依赖都满足一项：

- $\alpha \to \beta$是平凡的
- α 是 R 的超码

判断方式：直接判断给定的F中的函数依赖是否满足条件

分解方式 一定是无损分解：

$R = (A, B, C)$找到一条不满足BCFN要求的函数依赖$A \to B$ 提取一个关系(A,B),在R中删去B（A，C）。(尽量先从叶子上分解）

Dependency Preservation 依赖保持

> 在分解之后的关系中，能保证R上每一个函数依赖就是依赖保持的。
>
> $(F_1 \cup F_2 \ldots \cup F_n)^+ = F^+$

不能一定保持BCNF和依赖保持

⭐**3NF**

条件:

- 满足BCNF的一个条件
- $\beta - \alpha$ 包含在候选码中 （右边包含在一个候选码中）

分解方法 同时保证无损分解 以来保持

- 先找到正则覆盖
- 把正则覆盖中每一个函数依赖建一个表
- 如果候选码不包含在任何一张表中，添加一张表只包含候选码
- 删除冗余的表

多值依赖 应该不重要Jump

$\alpha \to\to \beta$和$\alpha \to\to R - \alpha - \beta$ 等价

分解方法：BCNF一样 注意这里的平凡是指：$\beta \subseteq \alpha \; or \; \beta \cup \alpha = R$

夏学期

*Lecture 8: Storage and File Structure

Storage Hierarchy: Primary storage （cache、 main memory） ,Secondary storage(flash disk)-online 、Tertiary storage (tape optional disks)--offline.

reliability:volatile storage 、 non-volatole storage

speed:cache\ main-memory \flash\ magnetic-disk\ optional storage \tape storgae

Performance Measures of Disks

Access time = Seek time + rotational latency

- access time = 发出读取请求到数据发出的时间
- seek time： 找到正确的磁道的时间，平均寻道时间是最坏情况的一半
- rotational latency time：旋转等待时间，平均时间是旋转一周的一半

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

RAID：只在一级五级选 以及写的性能好 但是内存消耗多

Buffer manager：LRU Random..

Lecture 9: Indexing and Hashing

索引类型:

- Ordered Indices： 按照 search key 顺序排列
- Hash Indices：物理存储离散

文件存储：Sequentially ordered file 也是根据一个 search key排列

Primary Index / clustering Index：索引和文件有相同的 Search key

- Index-swquential file 顺序索引文件 有主索引的文件

Secondary index：search key 不同 **不能用稀疏索引** 用桶存储指针

Dense Index File 稠密索引文件：文件中 search key 的所有值都能在索引中体现

Sparse Index 稀疏索引：只有部分取值在索引中体现

- 先找到最大的比 k 小的search key取值
- 顺序搜索 ——**所以不能是二级索引**
- 用于 **Good tradeoff** ： 每一个块中有多个数据，选取最小的作为整个块的索引。

> ***索引删除***

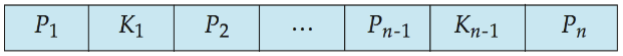
> *聚集索引:*
>
> *1. 唯一 直接删除*
>
> *2. 不唯一 主索引 指向下一个 。辅助索引 指向下一个bucket*
>
> *稀疏索引:*
>
> *1. 没有索引 不管*
>
> *2. 有索引 ， 修改为下一个*

> *插入索引*
>
> *聚集索引：单值直接添加 多值跳过*
>
> *稀疏索引：新block 建立新索引 原来block最小 修改索引*

🌟B+Tree Index Files

注意与ADS不同！！

结构

- 非叶节点有 $\lceil \frac{n}{2} \rceil$~$n$个孩子 （**指针**）
- 叶节点只能取$\lceil \frac{n-1}{2} \rceil$~$n-1$个值 （**与ADS区别**）
- 

节点结构
- 一个结点通常是一个Block
- **Fanout (扇出数)** 一个节点中指针的个数

  $n = \lfloor (4k(4096) - 4)/(4 + \text{一条数据大小}) \rfloor + 1$

查询：O(log n)

删除：1. 能借先借 2. 借不了合并 3.再考虑父亲(无论是先merge还是先借应当都可以 按照书本应该先借好一点)

🌟**树高**: 在这里应该理解成有几层

- **最大树高**：n 阶 K 个键值： $h = \lceil \log_{\lceil \frac{n}{2} \rceil} K \rceil$
- **最小树高**：$h = \lceil \log_n K \rceil$

- size大小的估计：也是两种极端情况

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

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

在计算树高和size时按照如下方法计算比较容易：

⚠️ **attention！！！**

> *有K个索引点，是n阶B+树，B+树叶子能放* $L_1 = \lceil \frac{n-1}{2} \rceil$ *到* $L_2 = n-1$ *个值 对于非叶子节点则是* $N_1 = \lceil \frac{n}{2} \rceil$ *到* $N_2 = n$ *个指针*
>
> *考虑**最高** 则是全部都放最少的情况 注意这里根节点可以只放2个指针*
>
> *于是就有:(下取整原因是，为了使得叶子尽可能多，如果又多出来的 新开一个节点就不满足条件了)*
>
> $x_0 = \lfloor \frac{K}{L_1} \rfloor, x_1 = \lfloor \frac{x_0}{N_1} \rfloor \ldots x_h = \lfloor \frac{x_{h-1}}{2} \rfloor = 1$ *如果要求总的节点就加起来就行：高度估计* $h = \lceil \log_{\lceil \frac{n}{2} \rceil} K \rceil \rightarrow h = \lfloor \log_{\lceil \frac{n}{2} \rceil} \frac{K}{2} \rfloor + 1$
>
> *对于高度**最小情况**相反*
>
> *下取整原因：叶子尽量少，多出来的节点又插不回去，于是就只能新开一个，并从其他节点凑出一个满足条件的*
>
> $h = \lceil \log_n K \rceil$

LSM Tree

内存中L_0达到阈值 就和L_1merge，以此类推

只有L_0在内存中 高阶都在硬盘中 每一层拆出多个小B+树（方便merge）

插入很快更新也很快，查找数据很麻烦

Lecture 10: Query Processing

语句执行顺序:parsing and translation(解析和翻译)、 optimization(优化)、Evaluation（执行）。

⭐Measure of query cost

两个消耗来源:

- seek :t_s
- transfer one block :t_t
    - read a block
    - write a block

总时间消耗 cost=$b \times t_t + S \times t_s$

Select operation

等值查找

File scan：不使用INDEX

A1（linear search）线性扫描：只需要seek一次 +所有的block的transfer *（一共有br个block）*

- 最坏情况：$b_r t_t + 1\, t_s$
- 查询值是主码找到即可停止：$\frac{b_r}{2} t_T + t_S$

A2 (binary search) 二分查找：文件必须顺序排列

- cost：$\lceil \log_2(b_r) \rceil \, (t_T + t_S)$

Index scan:使用INDEX、使用B+树 （这里默认 搜索对象就是**索引的searchkey**)

主索引搜查键值A3 (primary index, equality on key)：树根到树叶 做一次查找和transfer

- cost：$(h_i + 1)(t_T + t_S)$

主索引搜查非键值A4 (primary index, equality on nonkey):不是一个key，有许多相同结果需要再多搜索一些相同结果记录

- cost：$h_i(t_T + t_S) + t_S + t_T * b$ *(b是重复数据存在block数量)*

A5(secondary index) 二级索引

- 如果查的是个**键**（唯一结果）：与A3情况相同
- 当**不是键**时，查询到n个相同结果（可能在n个不同block上），于是最坏多做n次seek & transfer：$(h_i + n)(t_T + t_S)$：可能比较expensive 不如linear scan

- 还有一种储存方式是，叶子上指向存储指针的m个桶 桶中指针再指向n个块，$(h_i + m + n)(t_T + t_S)$

> ***比较查找与复合查找 （不重要）***
>
> ***区别在于遍历时 主索引不需要额外找块***
>
> *A6 (primary index, comparison, **基于主索引的比较**)：*
>
> - *A>v通过索引找到第一个满足的即可，从那里开始线性扫描*
> - *A<v 线性扫描直到找到A>=v 不用索引*
>
> *A7 (secondary index, comparison,**基于辅助索引的比较**)*
>
> - *A>v ,通过索引找到第一个满足条件的，再顺序扫描叶子*
> - *A<v 线性扫描所有叶子节点，直到找到A>=v*
>
> ***A8 (conjunctive selection using one index)：**先执行代价最小的条件，放回内存再执行下一个条件*
>
> ***A9 (conjunctive selection using composite index)：**正好复合索引和条件相同*
>
> ***A10 (conjunctive selection by intersection of identifiers)**如果部分有索引，可以单独做有索引的再交起来*
>
> ***A10 (disjunctive selection by union of identifiers)：**每一个条件先选择出来，再并起来直接使用线性扫描*

Sorting external merge sort

归并次数：$\lceil \log_{M-1}(b_r/M) \rceil$

总transfer传输次数：$2b_r\lceil \log_{M-1}(b_r/M) \rceil + b_r$

总seek操作：$2\lceil \frac{b_r}{M} \rceil + \lceil \frac{b_r}{b_b} \rceil (2\lceil \log_{M-1}(b_r/M) \rceil - 1)$

> ***解释**：*
>
> ***流程***
>
> - *第一轮传入M块数据，进行原地排序。（使用快排、堆排序等原地排序算法）*
> - *第二轮及后面每轮，合并M-1个有序序列。**Merge***
> - *注意，需要在memory块中预留一个区域存放排序好的有序序列，所以每轮合并后，有序序列的段数就少了M-1倍（预留出一个输出块的原因是：可以以块为单位写回，节省时间开销）*

cost of transfer

- *一次merge操作，选出M-1个页，以及一个临时页作为存储进行合并，于是每一轮Merge之后剩余原来的* $\frac{1}{(M-1)}$*,于是一共需要的轮数：*$\lceil \log_{M-1}(b_r/M) \rceil$
- *在每一轮的Merge中 所有数据都要被读出和写入，于是transfer开销* $2b_r$
- *假定最后一轮拍好之后不写回磁盘*
- *于是总的transfer次数为：*$2b_r\lceil \log_{M-1}(b_r/M) \rceil + b_r$ *(第一次的选出排序，之后所有轮的Merge 最后一轮只做一次transfer)*

cost of seek

- *第一次全部选出排序需要搜索次数：* $2\lceil \frac{b_r}{M} \rceil$ *读入和写入都需要seek 于是是两遍*
- *在Merge时每次需要读入一些参与Merge的块 设每次读入*b_b*块（这个块数是对于一个分组而言），于是一轮Merge就需要*$\lceil \frac{b_r}{b_b} \rceil$ *次读入和写入，一般来说*$b_r = 1$
- *但是*b_r*改变时* $M = \lfloor \frac{M}{b_b} \rfloor$ ***趟数****可能需要修改 具体看题吧如果标*估计不考 *（由于每一组一次读的块多了 那么能读入的组就少了）*
- *总计需要*$\lceil \frac{b_r}{b_b} \rceil (2\lceil \log_{M-1}(b_r/M) \rceil - 1)$*由于最后一轮不写*
- *最后结果为*$2\lceil \frac{b_r}{M} \rceil + \lceil \frac{b_r}{b_b} \rceil (2\lceil \log_{M-1}(b_r/M) \rceil - 1)$

Join operation

Nested-Loop Join：第一张表找一个元组 遍历第二张表

- 最坏情况，每一次只能读入这两个表的一个block
  - transfer：$b_r + n_r * b_s$
  - seek: $b_r + n_r$
- 最好情况：所有能全部读入
  - transfer：$b_r + b_s$
  - seek:2

Block Nested-Loop Join 两张表都先读入一个block，把这个block中的所有元组先进行自然连接

- 最坏情况 只能读两个
  - transfer：$b_r + b_r b_s$
  - seek:$2b_r$

- 最好情况直接全部读入
  - transfer：$b_r + b_s$
  - seek:2
- improvement：
  - 假设内存中能存放M块，第一张表一次读M-2块，里面的表一次读入一块，还有一块留给结果
  - transfer:$\lceil \frac{b_r}{M-2} \rceil * b_s + b_r$
  - seek:$2\lceil \frac{b_r}{M-2} \rceil$

Indexed Nested-Loop Join

- 原理：外层遍历tuple(每次读取一个block)，内层使用索引匹配（如B+树索引等)
- cost：$b_r \times (t_T + t_S) + n_r \times c$ *c是单次索引s的开销*

Merge Join

- 两张公共属性排好序的表，只需要遍历一遍
- cost：
  - transfers : $b_r + b_s$+排序（如果没排序）
  - seek $\lceil \frac{b_r}{b_b} \rceil + \lceil \frac{b_s}{b_b} \rceil$+排序（如果没排序）

hash Join:分别对每一个表建立一个哈希映射根据合并属性值进行分片，再通过哈希合并 **注意：两个n_h是不同的**

非递归：正常来说每一个都部分都在内存中有一个哈希块

$n = \lceil \frac{b_s}{M} \rceil$ *or* $n = \lceil \frac{b_s}{M} \rceil \times f(f = 1.2)$ 如果$n + 1 > M$则需要递归分区

内存空间中一个留给r 其他都是s

于是$b_b = \lfloor \frac{M}{n+1} \rfloor$

block transfer: $3(b_r + b_s) + 4n_h$

- 划分过程需要遍历表，包含读出和写回 $2 \times (b_r + b_s)$
- 匹配过程需要遍历表，只包含读出$b_r + b_s$
- 但是若Hash表非满，则至多制造出nh个非满块，涉及划分的写回和匹配的读出，作用于两个表，故共$4n_h$ (实际中nh一般很小，可以忽略不计)

seek times : $2(\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil) + 2n_h$

- 划分过程需要遍历表，假设每次放入内存块中b_b个，则读出写回共需$2(\lceil b_r/b_b\rceil+\lceil b_s/b_b\rceil)$次
- 匹配过程中直接取Hash块，共$2n_h$

若有递归划分 分成的部分数量大于可以容纳的块数时 进行递归分解

- block transfer:$2(b_r+b_s)\lceil\log_{M-1}(b_s)-1\rceil+b_r+b_s$
- seek times:$2(\lceil b_r/b_b\rceil+\lceil b_s/b_b\rceil)\lceil\log_{M-1}(b_s)-1\rceil+2n_h$

Materialization 实体化：构建操作树 前缀递归进行 要写入内存 开销大 用double buffer 加速 **基本是覆盖所有情况**

流水线Pipeline：不用写入内存 直接传给父操作，同时评估多个操作 **但并不是一直可以使用**

Lecture 11: Query Optimization

注意：在这里的操作都是byte为单位，4k bytes = 4096 bytes

基本步骤：

1. 获得等价逻辑表达式 2.获取一个查询计划（evaluation plan） 3. 选择最低预估消耗的

Transformation of Relational Expressions

Equivalence Rule(等价关系表达式)可以抄下来

1. $\sigma_{\theta1\wedge\theta2}(E)=\sigma_{\theta1}(\sigma_{\theta2}(E))$
2. $\sigma_{\theta1}(\sigma_{\theta2}(E))=\sigma_{\theta2}(\sigma_{\theta1}(E))$
3. 投影直接看最外层：$\Pi_{L1}(\Pi_{L2}(...(\Pi_{Ln}(E))...))=\Pi_{L1}(E)$
4. 选择可以变成笛卡尔积：
    i. $\sigma_\theta(E_1\times E_2)=E_1\bowtie_\theta E_2$
    ii. $\sigma_{\theta1}(E_1\bowtie_{\theta2}E_2)=E_1\bowtie_{\theta1\wedge\theta2}E_2$
5. 交换律：$E_1\bowtie_\theta E_2=E_2\bowtie_\theta E_1$
6. 结合率：$(E_1\bowtie E_2)\bowtie E_3=E_1\bowtie(E_2\bowtie E_3)$
    i. $\theta2$时只有2和3共有的属性：$(E_1\bowtie_{\theta1}E_2)\bowtie_{\theta_2\wedge\theta_3}E_3=E_1\bowtie_{\theta1\wedge\theta3}(E_2\bowtie_{\theta2}E_3)$
7. 选择优化先选择再连接
    i. (1) $\theta0$是只有E1有的属性：$\sigma_{\theta0}(E_1\bowtie_\theta E_2)=(\sigma_{\theta0}(E_1))\bowtie_\theta E_2$
    ii. (2) 1和2分别都是E1和E2独有的属性：$\sigma_{\theta1\wedge\theta2}(E_1\bowtie_\theta E_2)=(\sigma_{\theta1}(E_1))\bowtie_\theta(\sigma_{\theta2}(E_2))$
8. 如果θ 只包含$L1\cup L2$的属性，先投影后连接: $\Pi_{L_1\cup L_2}(E_1\bowtie_\theta E_2)=(\Pi_{L_1}(E_1))\bowtie_\theta(\Pi_{L_2}(E_2))$
9. 满足交换律$E_1\cup E_2=E_2\cup E_1, E_1\cap E_2=E_2\cap E_1$
10. 满足结合率：$(E_1\cup E_2)\cup E_3=E_1\cup(E_2\cup E_3),(E_1\cap E_2)\cap E_3=E_1\cap(E_2\cap E_3)$
11. 满足分配律：$\sigma_\theta(E_1-E_2)=\sigma_\theta(E_1)-\sigma_\theta(E_2)$
12. The projection operation distributes over union $\Pi L(E1\cup E2)=(\Pi L(E1))\cup(\Pi L(E2))$
13. 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

总结原则：

1. 先选择 再连接
2. 先投影 再连接
3. 连接时先连接小表

Statistical Information for Cost Estimation

- n_r:元组的数量
- b_r：块的数量
- l_r: 元组大小
- f_r:一个block能容纳turple的平均数量
- $V(A,r)$:在关系中属性A能取到同值的数量
- $b_r = \frac{n_r}{f_r}$

大小估计

注意操作顺序 有些情况不能死套下面的公式

操作	大小估计
$\sigma_{A=v}(r)$	均匀分布：$cost = \frac{n_r}{V(A,r)}$ 主键：1
$\sigma_{A\leq V}(r)$	$cost = 0$ v < min(A,r) $cost = n_r \frac{v-min(A,r)}{max(A,r)-min(A,r)}$
$\sigma_{\theta_1 \wedge \theta_2..\theta_n}(r)$	$cost = n_r \times \frac{s_1 \times s_2 ... s_n}{n_r^n}$
$\sigma_{\theta_1 \vee \theta_2..\theta_n}(r)$	$cost = n_r \times (1-(1-\frac{s_1}{n_r}) \times (1-\frac{s_2}{n_r})...)$
$\sigma_{\neg\theta}(r)$	$cost = n_r - size(\sigma_\theta(r))$
$r \times s$ or $r \bowtie s, R \cap S = \phi$	$cost = n_r \times n_s$
$r \bowtie s, R \cap S = key(R)$	$cost \leq n_s$
$r \bowtie s, R \cap S = foreign(s \to r)$	$cost = n_s$
$r \bowtie s$	$cost = \frac{n_r \times n_s}{max(V(A,r),V(A,s))}$ $cost = \frac{n_r \times n_s}{V(A,s))}$ R中每一个都参与
$\Pi_A(r)$	V(A,r)
${}_Ag_F(r)$	V(A,r)
左外连接 右外连接 全外连接	$size = size(r \bowtie s) + n_r$ $size = size(r \bowtie s) + n_s$ $size = size(r \bowtie s) + n_r + n_s$
$r \cup s$	$n_r + n_s$
$r \cap s$	$min(n_r, n_s)$
$r - s$	r

估计不同值的大小

操作	大小
$V(A,\sigma_{\theta_1}(r))$	1 特殊值筛选 特殊值数量 特殊集合筛选 V(A,r)s 筛选条件时A对于r的操作 $min(V(A,r),n_{\sigma_{\theta_1}(r)})$
$r \bowtie s$	$min(V(A_1,r)*V(A_2-A_1,s),V(A_1-A_2,r)*V(A_2,s),n_r \bowtie s)$

n个关系自然连接顺序： $\frac{2(n-1)!}{(n-1)!}$

Lecture12: Transactions

事务解决两个问题：1. 并发执行 2.系统错误硬件错误

Atomicity原子性： Either all operations of the transaction are properly reflected in the database or none are.

Consistency 一致性： Execution of a transaction in isolation preserves the consistency of the database. 单独执行事务保持一致性

Isolation 隔离性： Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

Durability持久性： After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

The **recovery-management component** of a database system implements the support for **atomicity** and **durability.**

active：活跃状态，事务正在执行

partially committed：最后一句被执行完，但是结果还是存储再buffer

Failed:事务执行失败

Aborted：rollback 之后会终端，两个选择：重新开始事务或者杀死事务

committed：完全提交，写入内存

Serial schedule 串行调度 保证数据一致性

concurrent schedule 并行调度

Serializability 可串行化

conflicting instructions:两个冲突操作不可以换位置，操作不冲突才可以交换次序（对于同一个对象 只有read read 无冲突）

conflict equivalent：通过交换无冲突指令 使得前后两种等价

conflict serializable：如果能和**串行调度冲突等价**

Recoverable Schedules：T_i读取了T_j之前写过的数据，需要保证T_j先提交（简单说：只能读已经提交了的数据） 否则如果T_j回滚 那么T_i读取的是错误数据。

Cascading Rollbacks：一个事务回滚导致一系列事务回滚

Cascadeless Schedules：避免联机回滚：保证事务是**可恢复调度**，就是写只有再提交之后才能被读

⭐**判断可串行化**

Precedence Graph前驱图

两个冲突的事务T_i, T_j，T_i先发访问这个冲突数据，则图中有一条 $T_i \to T_j$ 的边

如果是无环图则是冲突可串行化

通过拓扑排序，可以找到一个串行化方案

Lecture 13: Concurrency Control

Lock-Based Protocols

Exclusive(X) lock-x 表示事务可读可写 写锁：写数据加x锁

Shared（S）lock-s 表示只能读 读锁 读数据：加S锁

- 可以同时有任意多的事务持有S锁，但是只要有X锁就不能再持有这个数据项的锁
- 当无法获取某个数据项锁时，只有这个数据项的其他锁释放，才能获取

Dead Lock：两个事务锁互相等待，导致无法进行

Starvation：一个事务申请X锁，其他事务都在申请S锁

⭐**The Tow-Phase Locking Protocol**

一定能保证冲突可串行化 (proof :反证法)

- Phase 1：**Growing Phase** 只能获取锁，不能释放
- Phase 2：**shrinking Phase** 只能释放锁，不能申请锁

无法保证解决死锁问题 联级回滚也有可能出现

Strict two-phase locking

所有事务必须一致保持其X锁，直到commit abort 才释放

（保证了 我写完数据提交之后 才能读）**避免casecading roll-back 无法解决死锁**

Rigorous two-phase locking：事务提交之前不能释放任何锁

> *<以下应该不重要>*
>
> ***Lock Conbersions 锁转换 也能保证可串行化***
>
> - *第一阶段：获取锁，也可以将S锁升级为X锁*
> - *第二阶段：删除锁，也可以将X锁降级为S锁*
>
> ***Lock Manager***
>
> *能够接收事务的锁请求，并且对其进行回复*
>
> *锁存储内存中一个用数据项名称的哈希表中*
>
> *深色框代表被上锁，浅色代表在等待*
>
> ***Graph-Based Protocols按照图的顺序获取锁***
>
> - *只能获取X锁*
> - *第一个锁没有任何要求 之后的锁必须在指向它的所有锁获取之后才能获取*
> - *任何时间解锁，但是不能重复获取锁*
>
> *优势：能够**保证冲突可串行化**并且**防止死锁***
>
> *缺点：并不能保证可恢复和不联级回滚*
>
> ***Timestamp-Based Protocols***

> *W-timestamp:最大的写的时间戳 R-timestamp:最大的读的时间戳*
>
> *如果需要读取：TS<W 拒绝 TS>W 读 并修改R*
>
> *如果需要写：TS<R 拒绝 TS<W 拒绝*

Multiple Granularity 多粒度

- Intention-shared (IS，共享型意向锁)：存在某一个后代存在S锁
- Intention-exclusive(IX,排他意向锁)：后代存在X锁
- ShareShared and intention-exclusive (SIX，共享排它型意向锁):SIX=S+IX 后代都被加S锁 至少一个加X锁

	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

Deadlock Handling

prevention

- 拿到所有锁才能执行
- 使用偏序加锁
- **wait-die**：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**：老的事务强制让新的事务回滚而不等待其释放，新的事务等待。
- **Timeout-Based Schemes** 只等待一段时间，过了时间就回滚

Detection

wait-for graph：如果事务 **i 需要 j 释放一个数据项**，则在图中 画一条点 i 到点 j 的有向边，如果图中有环，说明系统存在一个死锁

Recovery

- 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
- **不能回滚太多 会导致饥荒发生**

Lecture 14: Recovery System

故障类型:

- Transaction failure：逻辑错误和系统错误（deadlock）
- system crash：系统崩溃的错误 断电等
- Disk failure：磁盘问题

Log-Based Recovery

> *事务开始* $< T_i, start >$
>
> *事务更新* $< T_i, x, v_{old}, v_{new} >$
>
> *事务提交*$< T_i, commit >$
>
> *事务回滚*$< T_i, abort >$

事务提交：将提交的日志写入stable储存空间时

数据库修改:

- Deferred Database Modification：提交之后才修改数据库
- Immediate Database Modification：活跃期间修改数据库

先做Undo 再做Redo

checkPoint：$< checkpointL >$:L是现在仍然活跃的事务，检查点之前的事务就可以不管了，只管后续的操作。

恢复算法

1. 初始化undo和redolist
2. 从尾部开始向前遍历 直到一个checkpoint
    1. 如果是提交日志 添加事务到redo-list
    2. 如果是开始 并且没有提交 则添加到undo-list
    3. 如果是中断 则添加到undo-list（应该理解为再做一遍undo 也就是忽略了之前undo的日志）
3. 如果对于checkpoint中活跃的事务并且不在redolist的话，则添加到undo-list
4. 倒叙遍历 undo 在 undolist中的事务
5. 再顺序遍历，redo 在redolist中的事务

> *Buffer rule*
>
> 1. *日志需要存储在稳定储存空间中*
> 2. *事务提交的标志是，提交事务的日志被写入稳定储存*
> 3. *只要提交日志被写入，之前的所有日志也应当被写入、*
> 4. *WAL 日志优先写入*

***Fuzzy checkpoint**:

为了减少在checkpoint时系统的暂停，于是不把修改的数据写入磁盘，只写入日志。同时将缓冲区的脏页记录，在其他时间抽空更新。同时需要维护一个last_checkpoint指针，直线最后一个安全的checkpoint。于是之后redo时就从lastcheckpoint开始。

***Logical undo logging**

```
<Ti,Oi,operation-begin>
//物理日志
<Ti,Oi,operation-end,U> //U是undo时的逻辑操作
<T0,O1,operation-begin>
<T0,C,700,600>
<T0,O1,operation-end,(C+100)>
...//c被修改成300
<T0,C,300，400>
<T0,O1,operation-abort>
<T0,abort>
```

ARIES Recovery Algorithm

LSN log swquence number:日志编号

Page LSN:每一页中最后一个影响这个页的LSN

Log Record：LSN+ TransID+ PrevLSN（同事务的上一条LSN）

Dirty Table：记录脏页

- PageLSN
- **RecLSN**：这个序列之前的操作都已经被写入磁盘了，也就是从这开始修改这个页。

checkPoint：

- Dirty table

- Active transactions: 活跃的事务，记录LastLSN，也就是最后一个操作的LSN

UndoList：记录 事务名称 和这个事务 的LastLSN

算法过程：

分析阶段 ANalysis pass：

1. **找到最后一个完整的checkpoint**
    1. 找到RedoLSN 是checkpoint 中dirtypage中最小的RecLSN
    2. undo-list 就是checkpoint中的事务（activite transactions）
    3. 找到所有的undolist中事务的last LSN
2. **从checkpoint正向扫描**
    1. 遇到不再undolist的就加入 提交了就删除
    2. 如果找到了更新的操作，需要更新脏页表 ，在其中只改LastLSN，不在需要添加并都设置成这个LSN

Redo Pass：

1. 从 RedoLSN开始扫描，遇到不在脏页表中的或者小于RecLSN的就跳过，其他重做
2. 从磁盘中读出这一页，如果磁盘中的PageLSN大 也跳过

UndoPass：反向扫描 重做 完成时需要写abort 并删除undolist