DataBase 期末复习笔记 2022

Karry

只有做事情，才能克服恐惧

原作（2022）：Karry
简注与增补（2024）：MountMist

【题目分析 - 题目很简单】

一、选择题（5 * 2 = 10 分 4 选 1）

二、简答题（4 * 5 = 20 分）

三、分析题（2 * 10 = 20 分）：

四、查询题（5 * 6 = 30 分）

五、设计题（2 * 5 = 10 分）。

六、优化题（10 分）

【复习规划】

基本上所有大题的知识点都已经全部给出，因此采用如下复习计划

1. 一定要把提及的知识点要全部理解清楚，梳理出相关的练习题
2. 在梳理大题的同时要把一些琐碎的知识点给拿出来

两步复习：

Step 1. 把张天庆老师复习的内容全部过一遍

Step 2. 和透露的题目类型对接起来 整理形成 复习练习题.md

Step 3. 刷试卷题目 + 刷课后习题

DataBase 期末复习笔记 2022

1 关系模式

1.1 关系数据库的结构

> 基本概念辨析

1. 关系数据库由表的集合构成，每张表被赋予一个唯一的名称。
2. 关系：被用来指 表
3. 元组：被用来指 表中的行
4. 属性：被用来指 表中的列
5. 关系实例：指代一个关系的特定实例，关系实例包含一组特定的行 值的概念
6. 关系模式：数据库关系设计的一个模板 type 的概念

   上面两个概念辨析很简单，关系模式就像 int

   关系实例就像 1 这个整数

7. 域：对于关系的每个属性都存在一个允许取值的集合（定义域）

   域要有原子性，即域中的每个元素不可再分，对于电话号码这个属性

   1. 如果说每个元组的电话号码是一个集合，电话号码以一个个集合作为域的元素的话，肯定不是原子性的

   2. 如果书每个元组的电话号码只有一个，那么我们也可以蒋电话号码拆分为不同的构成，这个时候就不能说他是原子性的。但是如果定义说电话号码就是不可以再分的，那么域就满足原子性的了。

8. 空值：表示值未知或不存在

1.2 码

> 一个元组的所有属性值必须能够唯一标识元组，也就是说一个表（关系）中不能由两个元组在所有属性上取值完全相同。
>
> 超码有超出的意思，最小的超码就是候选码，候选码中最有代表性的叫主码

1. 超码：一个或者多个属性的集合，将这些属性组合在一起可以允许我们在一个关系中唯一地标识出一个元组。 候选码就是键，超码是以键为子集的集合
2. 候选码：一个关系中可能由很多超码，但是我们只关心任意真子集都不是超码的最小超码，这就是候选码。

   主属性是指包含在任何一个候选码中的属性

3. 主码：数据库设计者选中的作为在一个关系中区分不同元组的主要方式的候选码。
4. 外码：关系 r1 中的每个元组对 A 的取值也必须是关系 r2 中某个元组对 B 的取值，A 就是 r1 引用 r2 的外码。 r1 是引用关系，r2是被引用关系。

【一个例子】就拿两个表来说

r1 : student(ID, name, dept_name)

r2 : dept(ID, dept_name)

对于 r1 这个关系：

1. ID 明显是超码，{ID, name, dept_name}也是超码，但是 name 就不是超码
2. 而以上超码中只有 ID 是候选码，因为只有其满足最小超码的概念，但是这并意味着一个关系只有一个候选码，候选码一定是超码。
3. 数据库设计者在设计的时候把 ID 这个最具代表性的候选码作为了主码，这是人为规定的。主码一定是候选码。
4. r1 的 dept_name 是一个外码，它引用了 r2 中的 dept_name
5. 外码约束比较特殊，要求选取的外码必须是被引用关系的主码

1.3 完整性约束

1. 实体完整性约束：关系的主码中的属性值不能为空值
2. 参照完整性约束：引用关系中的任意元组在指定属性上出现的取值也必然出现在被引用关系中至少一个元组的指定属性上。

2 基本 SQL

> 主要就是用于写出 SQL 语句，基础知识 + 后续练习写语句

2.1 SQL 数据定义

数据库中的关系集合使用数据定义语言（Data-Definition Language，DDL）定义的。DDL 不仅能够定义关系的集合，还能够定义有关每个关系的信息（各种各样的信息）

2.1.1 基本类型

char(n) 固定长度为 n 的字符串； varchar(n) 最大长度为 n 的可变长字符串

int smallint numeric(p, d) p 位，小数点后 d 位

float(n) 精度至少为 n 的浮点数

2.2 对表操作的基本语句 —— 必考点

2.2.1 创建表

```
【template】
creat table r
    (A1 D1,
     A2 D2,
     ...
     An Dn,
     <完整性约束1>,
     <完整性约束...>)
【case】
creat table instructor
    (ID varchar(20) not null,
     name varchar(20),
     salary numeric(8, 2),
     primary key(ID),
     foreign key(depat_name) references department)
```

说明：

1. 约束有三种：主码、外码、not null
2. 外码可以显式列出被引用表中的被引用属性，可以不列出

2.2.2 删除表

```
drop table r # 十分简单
delete table r # 尽管可以把所有的元组都删除掉，但是表仍然在
```

2.2.3 修改表

```
alter table r add A D # 在表中新添加属性 A 和 D
alter table r drop A # 在表中删除属性 A
```

2.3 查询

```
【查询书写顺序】
SELECT
FROM
WHERE
    GROUP BY
        HAVING
```

ORDER BY

【执行顺序】

FROM - WHERE - GROUP BY - HAVING - SELECT - ORDER BY

from:需要从哪个数据表检索数据

where:过滤表中数据的条件

group by:如何将上面过滤出的数据分组

having:对上面已经分组的数据进行过滤的条件

select:查看结果集中的哪个列，或列的计算结果

order by :按照什么样的顺序来查看返回的数据

2.3.1 基本查询结构

【基本架构】

```
select A1, A2, ... , An
from r1, r2, ... , rm
where P;
```

- select 子句用于列出查询结果中所需要的属性
- from 子句是在查询求职中需要访问的关系列表
- where 子句是作用在 from 子句中的关系的属性上的谓词

【理解方式】

Step 1. 先看 from： 为 from 子句中列出的关系产生笛卡尔积

Step 2. 在 Step 1 的基础上：应用 where 子句指定的谓词做筛选，得到指定表

Step 3. 在 Step 2 的基础上：对于指定表中所有元组，输出 select 子句中指定的属性

2.3.2 更名 AS

【基本用法】

as 可以用于 select 中更改属性的名称 ： 进而使表示更加直观

也可以用于 from 中更改表的名称 ： 进而完成表内的比较

【例子】

找出满足下面条件的所有教师的姓名，他们比 Biology 系教师的最低工资要高

```
select distinct T.name
from instructo as T, instructor as S
where T.salary > S.salary and S.dept_name = 'Biology'
```

2.3.3 字符串运算

【运算方法】

1. 比较是否相等
2. 函数转化，比如 upper(s) 转化为大写，lower(s)转化为小写
3. 比较是否相似：

    百分号(%)：% 匹配任意字串

    下划线(_)：_ 匹配任意一个字符

如果你想要保证字符串中有 % 或者 _ 那就需要使用转义字符：\% 以及 _

【例子】

```
where name like '%kai%' # 找到名字中包含 kai 的所有
```

2.3.4 排序 order by

【语法规范】

1. 是在 where 后面加入的一行
2. order by salary desc, name asc

    desc 是降序

    asc 是升序

    可以有多个属性，在第一个属性相同时比较第二个属性

2.3.5 集合运算

【并】 union 两个 select 出来的结果之间进行并

```
(select course_id
 from section
 where semester = 'Fall' and year = 2017)
union (all) union 会自动去重 如果想要保留所有重复项那就要加上 all
                  重复的会被加起来
(select course_id
 from section
 where semester = 'Spring' and year = 2018)
```

【交】 intersect 两个 select 出来的结果之间进行交

```
(select course_id
 from section
 where semester = 'Fall' and year = 2017)
intersect (all) intersect 会自动去重 如果想要保留所有重复项那就要加上 all
                          重复的会保留较大的那个集合中的数目
(select course_id
 from section
 where semester = 'Spring' and year = 2018)
```

【差】except

重复的会进行数量上的直接减(同样也是会有 all)

2.3.6 空值

空值不能和任何其他 type 的值进行运算（包括算术运算和逻辑运算），或者说运算出来的结果是 unknown

```
where salary is null
```

而不能是

```
where salary = null
```

‘=’ 是明显的数值运算

唯有"is null"这样的特定查询才能找到字段值是null的数据
而">3 or <=3"这样的数值查询是无法找到的

2.3.7 聚集函数

> 聚集的含义很清晰，说白了就是把一大堆数据融汇聚合在一起

定义是以值集（集合或多重集合）为输入并返回单个值的函数。SQL 提供了五个标准的固有聚集函数：

- avg | min | max | sum | count

用法可以是
SELECT avg(GPA)
FROM Student

其中 sum 和 avg 的输入必须是数字集，其他的可以在非数字几何上

基本用法：

- 可以在属性前面加 distinct 去重，也可以省略不去重。一定要仔细审题！
- count(*) 不能使用 distinct

count（*）是统计的所查询范围中的条目次数

分组聚集 —— 找到每个系得平均工资

- 必须采用分组 'group by 系别' 才可以得到正确结果
- 只有出现在 group by 中的属性 才能再次出现在 select 中，这个逻辑是很清晰的，要不值到底选哪一个呢？
- group by 的每一个组的含义要捋捋清楚

having 子句 —— 对 group by 后构成的每个组再做限制，并非针对单个元组 | 只对教师平均工资超过 42000 美元得那些系感兴趣。

处理空值 —— 除了 count（*）之外所有的聚集函数都忽略其输入集合中的空值

2.3.8 嵌套子查询

where 语句里面的条件还嵌套另外一个子查询

其中比较特殊的符号

- some：指某个 >some() 至少比一个大
- all：指所有 >all() 比所有的都大

2.3.9 测试空关系 —— 用于此类查找：找出选修了 Biology 系开设所有课程的学生。

既要查询 Biology 系开始的所有课程，又要查询所有学生所选的课

exist 是测试空集的好方法（非空返回 true），not exist 和其含义相反

```
select S.ID, S.name
from select student as S
where not exist ((查询 Biology 开始的所有课程) except (查询每个学生所选的课))
```

比较好理解，如果 Biology 开设的所有课程集合 除去某个学生所选课的集合还有剩余的话，那肯定就说明这个学生没有选所有的课

2.3.10 重复元组存在性测试 —— 找出在 2017 年最多开设一次的所有课程

unique 检测重复性，如果没有重复则返回 true 否则返回 false

```
select T.course_id
from course as T
where unique(select R.course_id
             from section as R
             where T.course_id = R.course_id and year = '2017')
```

注意：这个刚开始理解的时候好像不太好搞，但是掌握一个法则 —— 先用外面的查询定下来一张表的数据，然后这张表数据做子查询，这种思维是对的 或者是说按照元组的思维，毕竟 select 就是要选出某些行，这些行怎么选就是要一个个看

2.3.11 from 子句中的子查询 —— 找出所有教师工资总额最大的系

改变查询表的对象，先要把对象调整为每个系 + 每个系的工资总额，然后再在这个表中查询最大工资总额

2.3.12 with 查询 —— 嵌套查询的福音

> wtih 查询必须会，以后但凡用到嵌套查询，都直接采用 with 查询

说白了就是建立一个临时关系：找到工资总额大于所有系平均工资总额的系

明显，这个地方就已经有两层子查询了：要先查到所有系得工资总额，再在此基础上查到所有系得平均工资，如果直接用嵌套子查询的画写起来比较困难，所以我们直接用 with

第一个子关系：

```
with dept_total(dept_name, value) as
    (select dept_name, sum(salary)
     from instructor
     group by dept_name)
```

第二个子关系：

```
with dept_total_avg(value) as
    (select avg(value)
     from dept_total)
```

最终查询：

```
select dept_name
from dept_total, dept_total_avg
where dept_total.value > dept_total_avg.value
```

2.3.13 标量子查询

标量子查询是指：只返回一个包含单个属性的元组的查询，就好像一个标量，因此它可以出现在查询语句的任何一个地方。（但是本质上仍然是关系）

2.3.14 不带 from 的查询 *

```
(select count(*) from teachers) / (select count(*) from
instructor);
/* 明显是有问题的 因为你都不在前面 select 一下 它怎么 show 出来呢？但是有的系统的确支持 */
/* 为了避免出现问题 可以采用 SQL 默认的一个中转站 dual 来进行 select * from dual
```

2.4 修改 增删改（查）

2.4.1 删除

```
delete from r // r 表示关系
where P // P 表示条件
```

2.4.2 增

```
insert into r (r 中的某些字段)
        values (值) // 一一对应，注意 not null 约束
```

values 可以直接通过select进行构造再insert
如 insert into apply
...> SELECT sID,'Carnegie Mellon','CS',NULL
...> FROM student
...> WHERE sID not in (select sID from apply)
表示未报考的同学调剂去CM大学CS专业

2.4.3 更新

```
update r
set 更新前字段 = 更新后的值
where
/* 有的时候可能会好几次，所以要注意运用顺序 当然也可以采用如下方法 */
update r
set 更新前字段 = case
                    when pred1 then result1
                    when pred2 then result2
                    when pred3 then result3
                    ...
                end
```

3 中级 SQL

> 主要是知识应用 写语句不重要

3.1 连接表达式

连接类型：

- inner join
- left outer join
- right outer join
- full outer join

未指明的join指的是inner join

连接条件：

- natural
- on ...
- using <属性s>

自然连接要求所有相同属性都取值相等的才连接
而inner join using 则只是要求using s中的属性s取值相等

3.1.1 自然连接 natural join

定义：与两个关系的笛卡儿积不同，自然连接只考虑在两个关系的模式中都出现的哪些属性上取值相同的元组对，而笛卡儿积将第一个关系的每个元组与第二个关系的每个元组都进行串联。

例子：

```
select name, course_id
from student natural join takes
```

自然连接是幂等的

这样就会直接匹配到 student 和 takes 相同属性 ID 中有相同值的那些元组进行连接，效率更高

自然连接的实质是
1进行笛卡尔积
2仅挑选相同属性相等的元组
3消除两个表中的重复属性

问题与解决方式：

如果 A 关系和 B 关系中有两个相同的属性，就必须保证两个相同的属性值同时相等才能进行连接，这有时会产生错误，所以自然而然需要引入带条件的连接 join ... using

```
select name, title
from (student natural join takes) join coures using (course_id)
```

这样就保证说尽管前一个关系和后一个关系在多个属性上相同，但是只使用 course_id 进行连接

3.1.2 连接条件

除了上面指定某一个属性进行连接以外，还可以给与条件进行连接

```
A join B on ... 条件
```

事实上，如果没有on的限制，join就是笛卡尔积

```
select *
from student join takes on student.ID = takes.ID
'''注意和 where 的区分，其实本质上没有区别，但是一般用 on 表示连接条件；用 where 表示筛选条件
```

3.1.3 外连接

上述自然连接本质是一种内连接会出现这样的问题，如果 students 中有学生根本没有选课，或者有些课根本没人选，那么自然连接之后就会造成这些无法匹配的元组被直接剔除掉，那么为了保留这些在内连接中丢失的元组，通过创建包含空值元组的方式进行的连接就是外连接

- 左外连接：只保留出现在左外连接之前的关系中的元组 left outer join
- 右外连接：只保留出现在右外连接之后的关系中的元组 right outer join
- 全外连接：保留出现在两个关系中的元组 full outer join

外连接允许存在悬挂元组，dangling tuple（悬挂元组）是指在关系数据库中的一个表中存在一个或多个引用其他表中不存在的元组
左外连接允许左表中的属性在右表中没有对应的也返回（悬挂元组来自左表）
右外同理，全外允许这两种返回

3.2 视图

用户使用数据库总归会有特殊的目的，而并不是想看到数据库中所有的内容，因此需要把 select 出来的数据重新利用起来，给到用户。自然想到的一种方式是存储下来编号，之后用的时候给到用户，但是如果数据库内容变了怎么办？

因此需要建立一种与查询之间拥有动态的关系事务：视图(view)

3.2.1 视图定义

冲突可串行化 < 视图可串行化 < 可串行化

视图可串行化：如果能为一组指令找到一个视图等价的串行操作方式，则称这组指令是视图可串行化的（见DBSP9.1 47页）

视图等价的条件是：
1 相同的初值：同一指令T读取了相同的初值
2 相同的依赖：中间读写的先后顺序要相同
3 相同的写回：最后写回的指令是相同的

即只保证第一个读的仍然是第一个读，最后一个写的还是最后一个写，中间的靠盲写的机制来保证。

'''就是把视图和查询绑定'''

```
create view view_name as <查询表达式>
```

【case】

```
create view faculty as
    select ID, name, dept_name
    from instructor
```

'''是不是突然想到了 with 语句， with 语句也是一种显示构造查询并重新定义的方式，但是注意视图一旦创建在显示删除之前就是一直可用的，但是由with定义的命名子查询对于定义它的查询来说只是本地可用，其他查询用不了'''

3.2.2 视图使用

说白了视图就是和数据库绑定的一种查询，因此所有子查询可以出现的地方都可以使用视图。在 from 中

注意视图始终是动态更新的，在时间点上和数据库同步

3.2.3 视图更新

对试图的更新本质是对底层数据库关系的更新，会出现两种问题：

1. 视图中只给出了部分属性，而底层数据库关系中则含有所有属性，向视图中插入数据时如果只输入部分属性值，无法满足本质关系的属性要求
2. 视图来自于两个关系，仅插入部分属性值，会造成完全错位的情况

因此必须要满足以下三个条件才有可能直接对视图实现更新

1. from 子句中只有一个数据库关系
2. select 子句中只包含关系的属性名，不包含其他运算的中间结果
3. 没有出现在 select 子句中的任何属性都可以取 null 值
4. 查询不含有 group by 或 having 语句

还有一个小问题：更新完的数据由于不满足查询条件根本不会在视图中显示出来，只是在底部关系中显示了。

3.2.4 物化视图

某些数据库系统允许存储视图关系（能够像原本的关系一样构成一张可视化的表），并且保证：如果用于定义视图的实际关系发生改变，则视图也跟着修改以保持最新。这样的视图被称为物化视图。

保护物化视图一直在最新状态的过程称为物化视图保护，简称为视图保护。各种数据库系统的视图保护方式可能不同，有的定期更新，有的用的时候才更新

3.2.5 事务

`定义`说白了就是一组查询或更新语句的序列组成，构建起来的语句集合，这个事务执行完后可以提交或者撤销：

- commit work 提交，就会永久保存
- rollback work 撤销，就会回滚，覆盖操作

3.2.6 索引

`定义` 建立在关系表中某些属性上的一个数据结构，从而使得在查该属性时简化操作，不用再去扫描所有的元组了。

```
create index index_name on 表(表中的属性)
```

3.3 完整性约束

在 create table 的时候加上约束，从而保证表的合规性

3.3.1 单个关系上的约束

`1. not null` 非空约束，禁止对该属性插入空值，如果已经声明了某属性为主码，那么即使不显示地进行非空约束也是有该约束地

`2. unique` 唯一性约束，保证某个或者某些属性不能重复

`3. check` check(·) 可以对任何的谓词做检查，进而约束

3.3.2 参照完整性约束（外码约束）

```
foreign key(dept_name) references department(dept_name)
'''被指定的外码属性列必须声明为被引用关系的超码，要么直接是主码，要么是使用唯一性约束的属性'''
'''级联删除，如果在定义时表明可以级联删除，则在被引用关系中删掉某一个元组时，则在引用关系中和其关联的元组也会被级联删除'''
```

3.4 数据类型

还有一些最基本的 char varchar float

```
float numertic(12, 2) 12 位数, 2位小鼠
```

3.4.1 日期和时间类型

1. 日期：日历日期，包括年（四位）、月和月中的日
2. 时间：一天中的时间，时分秒，秒可以用 time(p) 来指定小数点后的位数
3. 时间戳：date + time

```
extract(filed from d) 从 date 或 time 或 timestamp 中提取出单独的域
fild:
    - year month day hour 等
还有一些宏变量:
    - current_time 当前时间（带有时区）
    - current_date 当前日期
    - localtime  当前的本地时间（不带时区）
    - localtimestamp （本地时间戳 不带时区）
```

3.4.2 也有类型转换和格式化函数

```
cast(ID as 新的类型)
```

3.4.3 缺省值可以直接通过 default 定义

3.4.4 大对象类型

把大对象（好几 MB 甚至 GB）的文件采用对象的方式

3.4.5 用户自定义类型

```
create type Name as (数据类型) final
```

3.5 授权 先掌握着这些，其余的内容后续可以再补充

3.5.1 权限的授予与收回

SQL 标准包括选择（select）、插入（insert）、更新（update）和删除（delete）

授权

```
grant <权限列表>
on <关系名或视图名>
to <用户/角色列表>
【case】
gtant update(budget)
on department
to rk, kl
```

收回

```
revoke <权限列表>
on <关系名或视图名>
to <用户/角色列表>
```

4 形式化查询语言

> 更加明确了 这章必定考 3 个题目
>
> 把所有的符号含义搞搞清楚 + 刷题

4.1 关系代数

五种基本关系代数运算是 并、差、笛卡尔积、选择、投影

4.1.1 选择 σ

$$\sigma_{dept\_name="phtsics" \wedge salary>9000}(instructor)$$

就是 select 运算，下标表示条件谓词

4.1.2 投影运算

$dept_name$这个属性可能根本就没有用，所以我们只选取剩下的几个属性，构成一个新的关系，使用投影运算，可以去除其他的所有属性同时删掉重复值

$$\Pi_{ID,name,salary/12}(instructor)$$

4.1.3 笛卡尔积运算

就是两个关系进行笛卡尔积

$$r = instructor \times teaches$$

4.1.4 连接运算

就是两个关系的连接（内/外连接）

$$r \bowtie_\theta s = \sigma_\theta(r \times s)$$

θ是条件

4.1.5 集合运算

并：$\cup$

交：$\cap$

差：$-$

4.1.6 赋值运算

$\leftarrow$ 可以进行顺序递接运算

4.1.7 更名运算

可以重复使用某个关系

$$\rho_{new\_name}(instructor)$$

4.1.8 除法运算 ÷

讲解：有关系 R 和关系 S

关系 R，包含 A、B、C 三个属性

A	B	C
a1	b1	c2
a2	b3	c7
a3	b4	c6
a1	b2	c1
a4	b6	c6
a2	b2	c3
a1	b2	c3

关系 S，包含 B、C、D 三个属性

B	C	D
b1	c2	d1
b2	c1	d1
b2	c3	d2

如何计算 R÷S 呢，首先我们引进"象集"的概念，具体意义看下面的陈述即可理解

关系 R 和关系 S 拥有共同的属性 B、C，R÷S 得到的属性值就是关系 R 包含而关系 S 不包含的属性，即 A 属性

在 R 关系中 A 属性的值可以取{ a1，a2，a3，a4 }

a1 值对应的象集为 { (b1,c2) , (b2,c1) , (b2,c3) }

a2 值对应的象集为 { (b3,c7) , (b2,c3) }

a3 值对应的象集为 { (b4,c6) }

a4 值对应的象集为 { (b6,c6) }

关系 S 在 B、C 上的投影为 { (b1,c2) , (b2,c1) , (b2,c3) }

只有 a1 值对应的象集包含关系 S 的投影集，所以只有 a1 应该包含在 A 属性中

所以 R÷S 为

A
a1

总之只需要注意两个点：

1. A ÷ B 得到的属性（字段）一定是 A 中有而 B 中没有的，因此如果 A B 字段完全相同那就没有任何 ÷ 的意义了

2. 对于那些 B 中有但 A 中没有的字段，我们完全不用 care ，如果 A 和 B 没有任何的公共字段那么 ÷ 出来就是空集

除法一般用于查询所有的 xx，比如查询选修了所有课程的同学的学号

4.2 数学运算符一样可以用在这

max min 这些都可以使用

5 E-R 模型与数据库设计

> 必考点：画 E-R 图 设计关系模式并给出查询
>
> 肯定还有其他零星的散点

上面我们都是给定了一个数据库然后来进行查询，但关键是数据库怎么来的，我们先学习采用 E-R 模型来进行设计

5.1 设计过程

5.1.1 设计阶段

1. 完整描述未来数据库用户的数据需求
2. 构建一种数据模型，并采用所选数据模型的概念将这些需求转化为数据库的概念模式。
3. 指明企业的功能需求 给出功能需求规格说明
4. 部署（逻辑设计阶段 + 物理设计阶段）
    - 逻辑设计阶段：把概念模式（E-R图）转化为关系模式
    - 物理设计阶段：如何在物理端（服务器）上进行部署

5.1.2 设计选择

避免两大问题：冗余和不完整

使用一张大表记录所有数据会产生冗余
坏处是会使得一致性的维护成本大大增加
好处是在某些时候能提高安全性（如备用钥匙）、访问速度（如cache）等
冗余会导致删除异常与更新异常

另外插入异常也是设计异常

将大表分解为小表可以解决以上问题
但是大表的优势在于查询时不需要连接
分解过程可以依据范式,但满足范式也不一定是好的分解

5.2 E-R 模型

5.2.1 实体集

矩形表示，有属性；

每个实体有特定的属性值，且它们具有的属性都相同

5.2.2 联系集

“ 实体集 ” 与 “ 实体 ” 类似于 “ 类 ” 与 “ 对象 ” 的概念
实体集具有的属性用椭圆表示，连到矩形上

一般而言每个实体集都需要有一个键，构成键的属性加下划线

实体集之间才有联系,菱形写在线上，如 [Bars]——<Sell>——[Beer]

用菱形表示，联系集上的属性用矩形 + 虚线（和实体集连接用实线）连起来，关系集两端的实体可以构成一个元组，所有的实体就构成了一张关系表。

参与联系集的实体集的数目是联系集的度

联系集也可以有多路，关联多个实体，但是整合时要保证符合已有联系。如酒客A喜欢去酒吧B喝啤酒C，但是酒吧B需要卖啤酒C

联系集自身也可以有属性，比如[Bars]——<Sell>——[Beer]的售卖关系可以有属性price，且该属性如果有多种选择，也可以另外构建为一个实体集。

5.3 复杂属性

INSTRUCTOR 这个表

属性 1. ID

属性 2. name (复合属性)

- first_name
- last_name

属性 2. address (多重复合属性)

- street
  - street_num
  - street_name
  - apt_num
- city

属性 3. [phone_num] (多值属性)

属性 4. sum(*) (派生属性)

子类（subclass）
用三角形表示，含义为“ is a ”

5.3.1 简单属性和复合属性

name 可以拆分为 first_name 和 last_name

5.3.2 单值和多值属性

phone_num 可能有多个值，是一个集合

5.3.3 派生属性

来自于另外的实体集，比如老师指导学生的数目

5.4 映射基数

表示规则

两个实体集之间可以有多种不同的关系

1. 1 的一方 有箭头，多的一方只有实线
2. 是否"全部"，如果 R 中每一个实体都要参与到联系中来，那么就是全部的（每一位学生都要有老师）反之则是部分的（每一位老师不一定都有学生）【参与性约束】
3. 基数限制，表示一个实体集中的实体可以对应另外一个实体集实体的数目【基数约束】

5.5 主码

区分实体集与关系集中每一个元素，对于实体集来说主码一般来说都是认为指定的，但是对于关系集来说，主码是根据实体集的主码构建的：

一对一 ： 任一主码

一对多 ： 多方主码

多对多 ： 两个主码交叉

弱实体集

弱实体集不含有键，
依赖于另一个强实体集的键加上该实体集的某属性才能确定弱实体

有的实体集本身已经存有一对多关系，比如每个学生选了很多门课，但是这个信息有的时候有些冗余，但是你去掉后就不能唯一标识了，所以我们定义了弱实体集：依赖于另一个实体集（标识性实体集）；使用标识性实体集的主码以及称为分辨符属性的额外属性来协助唯一地标识弱实体，就是说用课程的 id 以及选择的人的 id 来标示学生，我门把去除了选课信息的学生实体集称为弱实体集。

这样的强实体集称为支撑集↓

每个弱实体集必须依赖于另一个实体集

如球员是弱实体集，因为球员的名字和编号都有可能重复，加上“ 供职于 ”某球队（球队是强实体集），才能唯一确定该球员。
[[Players]]——<<Plays for>>——[Team]

弱实体集合用双边框的矩形来标示，关系用双边框的菱形来表示

5.6 将 E-R 图转换为关系模式（一定会考）

画出来 E-R 图（参考 P189 的画法）后，设计出表结构之前，需要得到完整的关系模式才能完成后续的设计。

5.6.1 具有复杂属性的强实体集的表示

对于复合属性来说，直接把复合属性拆开，组成一个个属性；

对于多值属性来说，直接再重新设定一个关系来进行多值表示。

5.6.2 弱实体集

一个实体没有全局ID时才使用弱实体集，应尽量少使用

因为弱实体集无法唯一标识每个实体，所以我们再进行关系模式转化时必须把弱实体集转化为强实体集

5.6.3 模式的合并

一对一的关系模式与一对多的关系模式可以直接进行合并，前者在哪个关系上合并都可以，后者需要在多者上进行合并。

5.7 拓展的 E-R 特性

5.7.1 特化和概化

特化 在实体集内部进行分组的过程称为特化，把人分成男生女生

概化 特化反过来，对某些实体进行汇总

5.7.2 属性继承

由特化和概化所产生的低层和高层实体的一个重要特性就是属性继承，他们会继承最底层属性的公共特征，并产生各自新的属性。

5.7.3 特化上的约束

- 全部特化或概化：每个高层实体必须属于一个低层实体集
- 部分特化或概化：一些高层实体可以不属于任何低层实体集

5.7.4 聚集

把一部分实体 + 联系聚集成一个抽象概念，更好表示

5.8 E-R 图设计的问题 （可能的选择题）

5.8.1 错误类型

1. 既然建立起来联系，就让联系集来表示实体集之间的联系，不要再用属性表示
2. 联系集中不要有实体集的主码

6 关系数据库设计

6.1 从问题出发，看不好的关系模式有什么缺陷

in_dep(ID, name, salary, dept_name, building, budget)

这个关系是把 instructor 和 department 联系在一起了，可以直接查询到每个老师的信息

但是有以下问题：

1. 对于每个 dept_name 都要重复一次 building 和 budget
2. 要想插入一个新的 dept_name 要有 ID 才行
3. 要想更改必须全部修改

因此我们后续就是要对这样不好的关系进行修改，使其满足一定的要求(范式)

分解 —— 对关系进行修改的方式

将大表分解为小表可以依据范式
有函数依赖时 使用 BC范式
有函数依赖+多值依赖时 使用 第四范式

范式强度:第四范式>BC范式>第三范式
范式越强,将大表分解得到的小表越多

上面的例子中我们把 in_dep 这个差的关系分解为 instructor 与 department 就可以解决上述提到的问题，后续解决问题的方式也都是分解：

- 无损分解：没有信息丢失的分解
- 有损分解：有信息丢失的分解

无损分解

将 R 分解为 R_1 和 R_2 必须满足：

$$\Pi_{R_1}(r(R)) \bowtie \Pi_{R_2}(r(R)) = r(R)$$

对于一般的分解，左包含于右。$\geqslant$ 而无损分解取到等号，即没有产生冗余的元组

分解后，自然连接回来和原本的关系是一模一样的

> 用函数依赖标示无损分解：把 $R(A,B,C,D)$ 分解为 $R_1(A,B,C)$ 和 $R_2(C,D)$ 则无损的条件是 R_1 和 R_2 的公共属性要么构成了 R_1 的超码，要么构成了 R_2 的超码

上述内容是无损分解的判定，可等价表述为，满足以下其中之一要求，就是无损分解：
R1∩R2→R1
R1∩R2→R2

范式

好的关系必须要满足的要求

6.2 函数依赖

该理论包括函数依赖于多值依赖，但多值依赖仅做了解

在关系 R，属性 A 能够唯一标识 B 那么 $A \to B$， B 函数依赖于 A，也称A函数决定B

定义（必须老老实实地背一遍）给定$r(R)$的一个实例，如果对于该实例中的所有元组对t_1和t_2，使得若 $t_1[\alpha] = t_2[\alpha]$，则 $t_1[\beta] = t_2[\beta]$ 也成立，那么我们就称该实例满足函数依赖$\alpha \to \beta$

α、β是属性，t是元组，属性α能够唯一确定属性β
假如属性α相同，则属性β也一定相同，例如学号与姓名

平凡的函数依赖：被所有关系都满足的函数依赖，一般来说如果 $\beta \subseteq \alpha$ 则 $\alpha \to \beta$ 的函数依赖就是平凡的。

实际语意是“整体确定部分”，所以是“平凡的”
另有“完全非平凡依赖”，指A→B且A∩B=空集

函数依赖集的闭包：F^+ 标示 F 的闭包，能够从给定集合 F 中推导出来的所有函数依赖的集合。

Attention：若有A→B，但不一定B→A。
若所有元组中都不存在两个元组在α上相等，也认为符合依赖条件(因为没有违反不相等的条件)

6.2.1 函数依赖集的闭包

对给定的函数依赖集 F 给出在模式上成立的所有函数依赖，计算 F^+ 是一个非常繁琐的过程。

阿姆斯特朗公理：

此方法可以推出基于当前函数依赖集的所有函数依赖
“正确且完备”（Sound and Complete）

- 自反律：若α为一个属性集且 $\beta \subseteq \alpha$ 则 $\alpha \to \beta$ 成立
- 增补率：若 $\alpha \to \beta$ 成立，且 γ 为一个属性集，则 $\gamma\alpha \to \gamma\beta$ 成立

- 传递率：若 $\alpha \to \beta$ 成立，且 $\beta \to \gamma$ 成立，则 $\alpha \to \gamma$ 成立

推论：合并分解规则
即两个函数依赖左部相同时，右部可以组合或分解
如：若有A→B且A→C，则有A→BC，反之也成立

6.2.2 属性集的闭包

属性才有闭包
闭包是属性集

令 α 为一个属性集。我们将函数依赖集 F 下被 α 函数决定的所有属性的集合称为 F 下 α 的闭包记为 α^+。说人话就是被 α 所决定的所有属性集合

α+称为正闭包，即不可能为空集的闭包，因为至少包含他自己

6.2.3 正则覆盖

计算闭包的过程是：
假设结果集是X
若B→C且
B在X中但C不在X中，
则将C也加入X中
重复以上步骤

如果所得闭包是全集，则该初始集是键

无关属性：去除函数依赖的一个属性，而不改变该函数依赖集的闭包，则称该属性是无关属性，去除了某个属性对函数依赖集没有实质性的影响。

正则覆盖：F 的正则覆盖 F_c 是这样的一个依赖集：F 逻辑蕴含 F_c 中的所有依赖，并且 F_c 逻辑蕴含 F 中的所有依赖。此外 F_c 具备如下性质：

- F_c 中任何函数依赖都不包含无关属性
- F_c 中每个函数依赖的左侧都是唯一的，不包括两个函数依赖存在相同的左侧

正则覆盖的算法

```
Step 1. 找到相同的左部后合并右部
Step 2. 删除无关属性
```

6.3 范式 （必考题）

范式描述的对象是表，依赖描述的对象是属性

6.3.1 Boyce-Codd 范式 （BCNF）

BCNF 消除了*基于函数依赖所能发现的所有冗余，虽然可能还保留着其它类型的冗余*

定义：关于函数依赖集 F 的关系模式 R 属于 BCNF 的条件是，对于 F^+ 中所有形如 $\alpha \to \beta$ 的函数依赖（其中 $\alpha \subseteq R$ 且 $\beta \subseteq R$，下面至少一项成立：

- $\alpha \to \beta$ 是平凡的函数依赖
- α 是模式 R 的一个超码

也就是说所有非平凡的函数依赖的左边都必须是 R 的一个超码

6.3.2 第三范式 （3NF）

定义：关于函数依赖集 F 的关系模式 R 属于 3NF 的条件是，对于 F^+ 中所有形如 $\alpha \to \beta$ 的函数依赖（其中 $\alpha \subseteq R$ 且 $\beta \subseteq R$，下面至少一项成立：

- $\alpha \to \beta$ 是平凡的函数依赖

- α 是模式 R 的一个超码
- $\beta - \alpha$ 中的每个属性 A 都被包含于 R 的一个候选码中

注意上面第三个条件并不是说单个候选码必须包含 $\beta - \alpha$ 中的所有属性；$\beta - \alpha$ 中的每个属性 A 可能被包含于不同的候选码中。

第三范式的等价描述是；
对于任意的A→B，
要么A是键，要么B是主属性

注意：凡是满足 BCNF 的一定满足 3NF

6.3.3 BCNF vs 3NF

3NF 的优点是：总可以在不牺牲无损性或依赖保持性的前提下得到 3NF 的设计，缺点是可能不得不用空值来表示数据项之间的某些可能有意义的联系，并且存在信息重复的问题。

核心目标是：

- BCNF
- 无损性
- 依赖保持性

无冗

无损

保依（可以降低要求）

BCNF与3NF都是无损的
BCNF保证无冗余但不保证函数依赖
3NF保证函数依赖但不保证无冗余

再次强调一次，BCNF 是基于函数依赖得到的范式，他也只能处理基于函数依赖所发现的冗余，所以并不是说 BCNF 就一定没有冗余了

6.3.4 【必考点】 范式检测算法

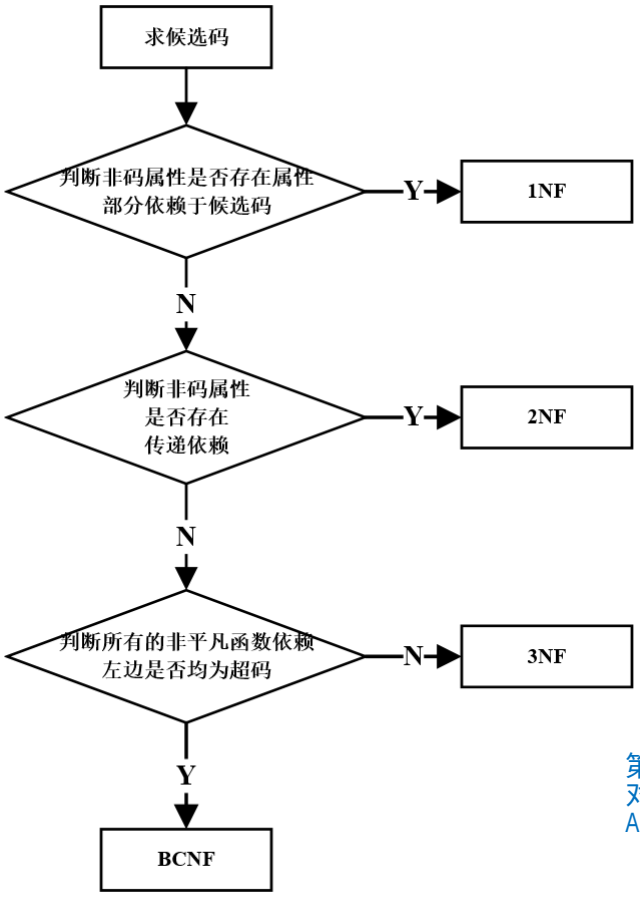
范式判断算法

多值依赖
A多值决定B，记作A→→B，
含义为属性A和属性B能够决定剩余的属性(?对吗?)
：不是，不能唯一确定，定义上是“存在”
同理，A→→B也可以推出A→→rest

当一个属性X的值确定时，另一个属性Y不仅仅是有一个值与之对应，而是有多个值与之对应。这些值的集合仅由X的值决定，与其他属性无关。

存在一对多的映射时，表中会出现多值依赖
如一个学生选了多门课，参加了多个社团
则学生多值决定课程/社团，而课程与社团之间独立

函数依赖也是一种多值依赖，
但多值依赖不一定是函数依赖



所有属性都依赖于它的键
即存在键就是第二范式

第四范式
对于R中所有的多值依赖A→→B，
A是一个键

【术语解释】

C D 均为非码属性

1. 部分依赖：如果 AB 是候选码，且 B 可以直接推出 C 那么就说 C 部分依赖于 AB

2. 传递依赖：如果 AB 是候选码 存在 AB 推出 C 并且 C 推出 D

一个例子： $R(A,B,C,D); F=\{B\to D, D\to B, AB\to C\}$

1. Step 1. 求候选码：

   先看属性在左右两边出现的情况

   L - 仅在左边出现 | LR - 左右两边都有 | R - 仅在右边出现

   L : {A} 首先怀疑 A 是候选码，但是发现什么都推不出来

   LR : {B, D} 那么怀疑 {A, B} {A, D} 是候选码，发现果然都是，就不再继续了

   因此候选码就是 {A,B} {A, D}

2. Step 2. 判断非码属性是否存在部分依赖：

非键属性只有 C

发现 C 只能由 AB 推出，所以是直接依赖，而非部分依赖

3. Step 3. 非键属性是否传递依赖：

   发现是直接导出

4. Step 4. 非平凡依赖是否为候选码：

   发现 B -> D 就不是，所以不是 BCNF 只是 3NF

一个练习： $R(A,B,C);F=\{A\to B,B\to A,A\to C\}$ 这个满足 BCNF

6.3.5 【必考点】 范式分解算法（分解为 BCNF 或者 3NF）

范式分解算法

一个例子：

$R(A,B,C,D,E,F);F=\{AE\to F,A\to B,BC\to D,CD\to A,CE\to F\}$

6.3.5.1 三范式分解

Step 1. 求候选码

先看属性在左右两边出现的情况

L - 仅在左边出现 | LR - 左右两边都有 | R - 仅在右边出现

L : {C, E}

首先怀疑CE 是候选码，但是发现不行

LR : {A,B,D}

从 LR 中选出一个和 CE 组合

ACE => 可以推出

BCE => 可以推出

DCE => 可以推出

一共有三个候选键

R : F

因此候选码就是 ACE BCE DCE

Step 2. 求函数依赖 F 的正则覆盖 F_c，发现该函数依赖 F 本身就是一个正则覆盖

Step 3. 把 F_c 都写成关系，并检查是否存在候选码，如果没有候选码，就补一个候选码作为一个关系

$$\rho=\{R_1(AEF),R_2(AB),R_3(BCD),R_4(CDA),R_5(CEF)\}$$

发现不存在候选码，所以补一个候选码作为关系即可

$$\rho^{3NF} = \{R_1(AEF), R_2(AB), R_3(BCD), R_4(CDA), R_5(CEF), R_6(ACE)\}$$

Step 4. 如果其中有关系包含在另外的关系当中，那么就要删去该关系。上述给出的分解没有被包含在其中的关系。

至此分解完毕！

6.3.5.2 BCNF 范式

有函数依赖的分解为BCNF，
有多值依赖的分解为第四范式

Step 1. 求候选码 和上面相同

Step 2. 求函数依赖 F 的正则覆盖 F_c，发现该函数依赖 F 本身就是一个正则覆盖

Step 3. 逐层分解

R(ABCDEF) F{AE->F, A->B, BC->D, CD->A, CE->F}

1.

因为 AEF 不是候选码，所以构成 R1(AEF)

同时R -> (ABCDE) F{A->B, BC->D, CD->A} 所有含 F 的都没了，因为 F 可以被推出

2.

因为 AB 不是候选码，所以构成 R2(AB)

同上 R -> (ACDE) F{CD -> A}

3.

因为 CDA 不是候选码 所以构成 R3(CDA)

同上 R -> (CDE) F{无}

R4 = (CDE) 肯定是一个候选码

综上得出：R 的分解 R1 R2 R3 R4

BCNF的分解方法.杨宁.ver
1 计算表R的键
2 重复以下步骤直至所有的函数依赖都满足BCNF
①选取违反BCNF的函数依赖A→B
②将表R分解为R1(A,B)与R2(A,rest)
//其中rest是R中除了A,B外的其他属性
③计算出R1与R2的函数依赖
④计算出R1与R2的键
3 去除不符合BCNF的中间产物

7 查询处理和查询优化

7.1 整体概述

查询处理是指从数据库中提取数据所涉及的一系列活动。

Step 1. 将用高层数据库语言表示的查询语句翻译为能在文件系统的物理层上使用的表达式，类似于编译器的语法分析器所作的工作。

Step 2. 查询优化，不是写查询语言人的义务，而是系统的责任

执行原语：带有“如何执行”注释的关系代数运算

查询执行计划：执行一个查询的原语操作序列，一个查询可能有多个查询执行计划

查询执行引擎：接受一个查询执行计划，执行该计划并把结果返回给查询

为了优化一个查询，查询优化器必须知晓每种运算的代价，尽管精确计算出代价是苦难的，但是可以粗略估计出。

7.2 查询代价的度量

使用从存储中传输的块数以及随机I/O访问数作为估计查询执行计划的代价的两个重要因素，这二者中的每一个都需要在磁盘存储器上进行磁盘寻道。

7.3 关系代数运算的执行 —— 十分难，但是老师没有提需要背暂时不掌握

7.4 表达式的执行

7.4.1 物化

把每个中间的运算结果都创建出来，用于下一层运算的执行，这种方法把运算结果写道磁盘再读，一方面存储压力增加，另一方面读取时间增加。

双缓冲是指利用两块缓冲区，其中一块用于连续执行算法，另一块用于写出结果，通过并行执行 CPU 活动与 I/O 活动，允许算法执行得更快。通过为输出缓冲区分配额外的块以及一次写出多个块，可以减少寻道的次数。

7.4.2 流水线

定义 将多个关系运算组合成一个运算的流水线实现，其中一个运算的结果将传送到流水线中的下一个运算。

优点

1. 消除了读和写的临时关系的待见，减少了查询执行代价；
2. 可以迅速开始产生查询的结果

执行方式

1. 需求驱动流水线：系统不停地向位于流水线顶端的运算发出需要元组的请求，每当一个运算收到需要元组的请求时，它就计算待返回的下一个元组并返回这些元组。

2. 生产者驱动流水线，各运算并不等待产生元组的请求，而是积极地生产元组。

使用生产者驱动流水线方式可以被看作将数据从一棵运算树的低层推上去的过程；而使用需求驱动流水线方式可被堪称从运算树的顶层将数据拉上来的过程。

7.5 查询优化 ** 必考点

给定一个关系代数表达式，查询优化器的任务是产生一个查询执行计划，该计划能计算出与给定表达式相同的结果，并且以代价最小的方式来产生结果。

查询优化的启发式方法：

- 尽早执行选择
- 今早执行投影

【例子 —— 就是最后一个答题的标准形势】

Find the names of all instructors in the Music department who have taught a course in 2017, along with the titles of the courses that they taught.

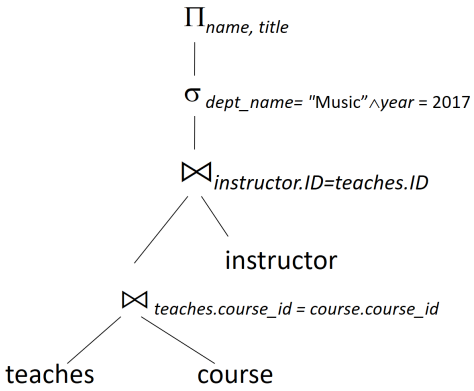
Step 1. 写出 Sql 查询

```
SELECT name, tiles
FROM instructors, teaches, course
WHERE instructors.id=teaches.id,
teaches.course_id=course.course_id, year=2017, dept_nam ='music'
```

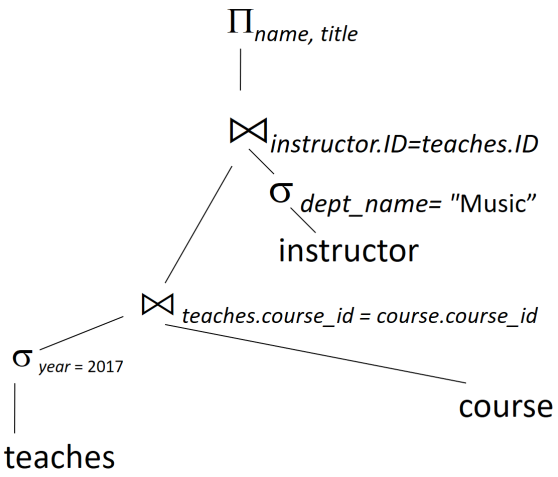
Step 2. 写出关系代数形式

$$\Pi_{name,title}(\sigma_{dept\_name="Music"\wedge year=2017}(instructor \bowtie_{instructor.ID=teaches.ID} (teaches \bowtie_{teaches.course\_id=course.course_id} (course))))$$

Step 3. 画出语法树

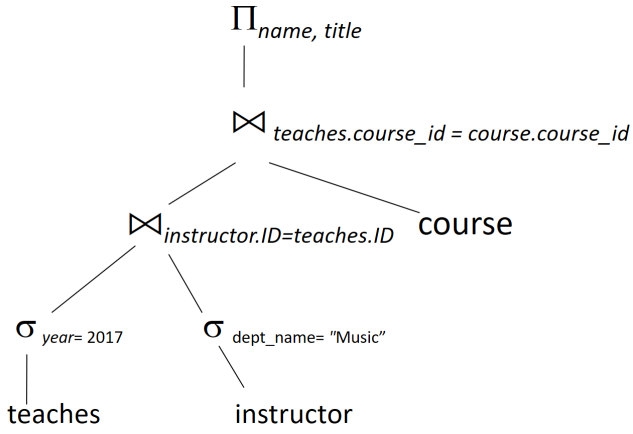


Step 4. 先把可以下移的选择下移

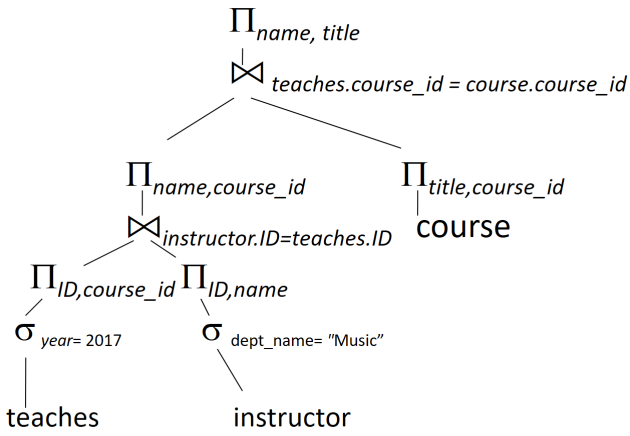


Step 5. 再把尽量轻量级的连接先进行了

> 这一步不一定简化



Step 6. 能早做的投影早做了



8 事务管理

8.1 基本概念

YN：事务是视为原子性单元的一系列操作

事务 是访问并可能更新各种数据项的一个程序执行单元。

ACID 特性 ：

1. 原子性（Atomicity）：事务所有的操作在数据库中要么全部正确反映出来，要么完全不反应 （要么全都执行，要么全都不执行）
2. 一致性（consistency）：以隔离方式执行事务（没有其他事务并发执行）以保持数据库的一致性 （当事务结束时，数据的状态要合乎逻辑，如“ 转账前后双方总额不变 ”）
3. 隔离性（isolation）：尽管多个事务可能并发执行，但是系统保证每个事务都感觉不到系统中有其他事务在并发地执行。让结果等效于各个事务串行运行后的结果
4. 持久性（durability）：在一个事务成功完成之后，它对数据库的改变必须是永久的，即使出现系统故障也是如此

使用日志及日志的回滚来保证原子性与持久性

8.2 可串行化

定义 衡量并发执行的性质，在并发执行的情况下，我们通过保证所执行的任何调度的效果都与没有任何并发执行的调度效果一样来确保数据库的一致性。

冲突可串行化

1. 冲突：如果 I 与 J 是不同事务在相同数据项上执行的操作，并且其中至少有一条指令是 write 操作，那么就说 I 与 J 这两个操作是冲突的

但冲突指令的顺序是不能交换的

2. 冲突等价：如果调度 S 可以经过一系列非冲突指令的交换而转换成调度 S' 则称 S 与 S' 是冲突等价的。
3. 冲突可串行化：如果一个调度 S 与一个串行调度是冲突等价的，则称调度 S 是冲突可串行化的（很好验证，把调度 S 中的事务串行起来即可）

冲突可串行化是可串行化的一个子集，不是冲突可串行化不一定是不可串行化

检验冲突可串行化的方法

通过交换非冲突的执行顺序寻找串行调度（让一个事务一次性做完）

1. 用定义，直接串行之后看结果，是否能够找到一个串行调度
2. 构件优先图，满足三个条件之一就表明 I 优先于 J，如果有环就不行
   - Tj 执行 read(Q) 之前 Ti 执行了 write(Q)
   - Tj 执行 write(Q) 之前 Ti 执行了 read(Q)
   - Tj 执行 write(Q) 之前 Tj 执行了 write(Q)

i优先于j就是说明i有一条边指向j
有边存在的条件是：
如果i中有一个操作0i与j中的一个操作0j冲突且0i在0j之前，则i有一条边指向j

可恢复调度 ： 对于每对事务 Ti 和 Tj，如果 Tj 读取了由 Ti 之前所写过的数据项，则 Ti 的提交 commit 操作应在 Tj 的 commit 之前（别人打了草稿你就超过来了，万一别人后面还改呢，你要等到它确定交卷后再交卷）

无级联调度 ： 因为单个事务失效导致的一系列事务回滚的现象称为级联回滚。无级联回滚是这样的情况：如果 Tj 读取了先前 Ti 所写的一个数据项，那么 Ti 的 commit 就一定要出现在 Tj 的读取前。（只有等到别人交卷了，我才抄，否则别人改我就得改）

8.3 事务的隔离性级别

可串行化是最强的隔离级，也是默认的隔离级

1. `Serializable` 可串行化：通常保证执行时可串行化的 000
2. `repeatable read` 可重复读：只允许读取已提交的数据，并进一步要求在一个事务两次读取一个数据项期间，其他事务不得更新数据项 001
3. `read committed` 已提交读：只允许读取已提交的数据，但不要求可重复读。 011
4. `read uncommitted` 未提交读： 允许读取未提交数据。 111

explain：数字表示会不会产生以下现象-脏读/不可重复读/幻觉

9 并发控制与恢复

> 为了确保事务的隔离性，系统必须对并发事务之间的交互加以控制

9.1 基于锁的协议

对公共数据项以互斥的访问方式进行访问，仅当一个事务当前持有一个数据项上的锁的情况下，该事务才能够访问数据项。

9.1.1 锁的定义

总共有两种锁：

锁越粗(封锁的节点靠近根节点)，并发度低，管理成本低
在数据库的树状结构中封锁一个节点就意味着也封锁了它的所有子节点
理论上来说需要在所有子节点都没有锁时候才能为该节点加锁，但是查询成本极高

1. 共享锁（shared-mode lock） 如果一个事务 Ti 获得了数据项 Q 上的共享模式锁，则 Ti 可以读 Q，但不能写 Q
2. 排他锁（Exclusive-mode lock） 如果一个事务 Ti 获得了数据项 Q 上的排他模式锁，则 Ti 可以读Q也可以写 Q

为了解决这个问题

“ 意图 ”是一个“ 性质 ”，可以与共享锁与排他锁兼容，即“ 意图共享锁 ”

COMPARE:
“ 共享锁 ”意味着为这个节点及其所有子节点加锁；
“ 意图共享锁 ”意味着这个节点的子节点中存在共享锁

执行 lock-S(Q) 指令来申请数据项 Q 上的共享锁，执行 unlock(Q) 指令来对数据项 Q 解锁（X 也是一样 表示排他锁）只有共享锁和共享锁是相容的，其他的都不相容。如果要访问一个数据，事务Ti必须首先给该数据加锁，如果该数据项已经被另外的事务加了不相同的锁，则在被其他事务所持有的所有不相容模式的锁被释放之前并发控制管理器不会授予该锁。Ti 只能等待

所以我们使用“ 意图锁 ”，为一个节点加锁就先为它的所有子节点加上意图锁，只有当意图锁没有产生冲突时，才能成功加锁。

加锁可能会导致死锁发生。所以要想避免不一致性同时避免死锁，就规定了封锁协议：系统中每一个事务都遵从的一组规则，规定何时加锁和解锁。

9.1.2 锁的赋予

如果刚开始一个事务对某个数据上了S锁，下一个事务企图上X锁，但是后面又有很多事务都期待上 S 锁，这种情况下 X 锁永远都赋予不了。

当一个事务 Ti 申请多一个数据项 Q 加特定模式 M 的锁时，设置条件：

1. 在 Q 上持有与 M 冲突模式的锁的其他事务是不存在的
2. 正在等待对 Q 加锁且先于 Ti 提出其锁申请的事务是不存在的

9.1.3 两阶段封锁协议

定义：

1. 增长阶段：一个事务可以获得锁，但不能释放任何锁
2. 缩减阶段：一个事务可以释放锁，但不能获得任何新锁

起初一个事务处于增长阶段。事务根据需要获得锁，一旦事务释放了一个锁，它就进入了缩减阶段，并且不能再发出加锁请求。

一旦开始释放锁就不能再申请锁了

两阶段封锁并不保证不会发生死锁，只能保证冲突可串行化

封锁点：一个事务得到了所有需要的资源（申请了所有需要的锁）
也即增长阶段与缩减阶段的分界点

封锁点时其他与该事务冲突的事务要么已经释放了锁，要么还在等待申请锁（封锁点有先后），就使得这些事务能够等价于按封锁点先后串行执行

普通协议是“你借了五本书可以看完一本还一本”，严格协议是“五本书必须一起还”

为了避免级联，提出两种替代方案

严格协议要求申请可以逐渐申请，但是所有锁必须一起释放，而不是逐渐释放

1. 严格两阶段封锁协议：不但要求封锁是两阶段的，而且要求事务所持有的所有排他模式锁必须在事务提交后方可释放。
2. 强两阶段封锁协议： 要求在事务提交之前保留所有的锁。

锁转换是指：S 锁和 X 锁的转换。 S => X 是升级，X => S 是降级

9.2 死锁处理

对死锁更为细致的定义：如果存在一个事务的集合，使得该集合中的每个事务都在等待该集合中的另一个事务，那么就是处于死锁状态（每一个都在等待）。

9.2.1 死锁预防

方法 1．对封锁请求进行排序，或要求同时获得所有的锁保证不会发生循环等待

- 在事务开始之前，通常很难预知哪些数据需要封锁
- 数据项利用率可能极低，许多数据项可能被封锁但却长时间不被使用

方法 2．使用抢占与事务回滚，若一个事务 Tj 所申请的锁被事务 Ti 持有，则授予 Ti 的锁可能通过回滚 Ti 而被抢占。但是这种回滚必须要按照一定的规则才行，要不就乱套了。

1. 等待-死亡，当事务 Ti 申请的数据当前被 Tj 持有时，仅当 Ti 比 Tj 老时才允许 Ti 等待，否则 Ti 回滚
2. 伤害-等待，当事务 Ti 申请的数据当前被 Tj 持有时，仅当 Ti 比 Tj 年轻时才允许 Ti 等待，否则 Tj 回滚 （Tj 被破坏掉）

当一个在等时不会让另一个也在等

9.2.2 锁超时

超时是简单而实际可行的方法

申请锁的事务至多等待一段指定的时间，若在那段时间内尚未授予给该事务，则称该事务超时，进而该事务回滚并重启。

9.2.3 死锁检测与恢复

如果你不能避免死锁，那就必须定期调用检查系统状态的算法以确定是否发生了死锁，如果发生死锁则系统必须试着从死锁中恢复。

9.2.3.1 死锁检测

使用等待图来表示资源索取情况，很简单，箭头的方向表明等待资源的方向 A => B 表示 A 等待 B。如果没有环就不会发生死锁

在现实系统中并不会使用该方法，因为事务（节点）数量太多了

9.2.3.2 从死锁中恢复

要想恢复，就必须执行以下三个动作：

1. 选择牺牲者：给定一组死锁的事务，必须决定回滚哪一个事务以打破死锁
2. 回滚
3. 饿死：可能有的事务一直在被回滚，直至饿死

9.3 多版本并发控制

定义 每个 write(Q) 操作创建 Q 的一个新版本，进行 read(Q) 操作时，并发控制管理器选择 Q 的一个版本进行读取。

9.4 故障分类

1. 事务故障：
    - 逻辑错误：事务由于某些内部情况而无法继续其正常执行，这样的内部情况诸如非法输入、找不到数据、溢出或超出资源限制
    - 系统错误：系统进入一种不良状态，其结果是事务无法继续其正常执行，但该事务可以在之后的某个时间重新执行
2. 系统崩溃：硬件故障，或者是数据库软件或操作系统的漏洞，导致易失性存储器内容的丢失，并使得事务处理停止。但非易失性存储器内容完好无损且没被破坏。
3. 磁盘故障：在数据传输操作中由于磁头损坏或故障造成磁盘块上的内容丢失。（必须备份）

恢复算法

1. 在正常事务处理中采取措施，保证存在足够的信息可用于故障恢复。
2. 故障发生后采取措施，将数据库内容恢复到某个保证数据库一致性、事务原子性及持久性的状态

9.5 恢复与原子性

为了保持原子性的目标，我们必须在修改数据库本身之前，首先向稳定存储器输出信息，描述要做的修改，将看到：这种信息能帮我们确保已提交事务所做的所有修改都反映到数据库中，如果执行修改的事务失败，我们还需要存储有关被修改的任意项的旧值信息。

9.5.1 日志记录

更新日志记录中含有以下表项：

1. 事务标识：执行 write 操作的事务的唯一标识
2. 数据项标识：是缩写数据项的唯一标识，通常是数据项在磁盘上的位置，包括该数据项所驻留的块的块标识 + 块内偏移量
3. 旧值：数据项写之前的值
4. 新值：数据项写之后的值

9.5.2 事务提交

提交后的事务的影响必须持久保存

当一个事务的 commit 日志记录被输出到稳定存储器后，我们就说这个事务提交了，此时所有更早的日志记录都已经被输出到稳定存储器。

9.5.3 使用日志来重做和撤销事务

脏数据是未提交事务更新的数据已经写回
脏页是内存页面已更新但是磁盘尚未更新

1. REDO 重做 ： 将事务 Ti 更新过的所有数据项的值都置成新值
2. UNDO 撤销 ： 将事务 Ti 更新过的所有数据项的值都恢复成旧值。

还没有 commit 的事务，恢复时就采用 UNDO 的方式

被 commit 的事务在恢复时采用 REDO 的方式

UNDO保证原子性，REDO保证持久性，
做了一半中断还未提交的就重新做，
做完且提交了但是没写回的要写回。
写回时要先写日志再写磁盘

9.5.4 检查点

说白了就是在所有事务的执行中打点，检查点之前的事务已经完成了，崩溃时就不再去过问，检查点之后的事务 commit 的需要 Undo ，没有 commit 的需要 REDO

9.6 恢复算法

9.6.1 事务回滚

正常操作下的事务回滚，会从后往前进行扫描并且处理

9.6.2 系统崩溃后的恢复 **

1. 重做阶段：重演所有事务的更新，并且得到 undo-list
2. 撤销阶段：对 undo-list 中的所有事务进行回滚，从尾端开始反向地扫描日志来执行回滚。up

已完成事务-已提交　未刷脏 redo
　　-已终止　未刷脏 redo
　　---->保证持久性

未完成事务-已刷脏 undo -->保证原子性

要解决原子性和持久性问题，分别有STEAL/NO-STEAL,FORCE/NO-FORCE两种策略
两两组合会产生四种恢复算法

数据库日志的三个性质
- 幂等性：一条日志记录无论执行一次或多次，得到的结果都是一致的。
  -例如：x=x+1不幂等；x=0幂等
  -物理日志满足幂等性；逻辑日志不满足
- 失败可重做性：一条日志执行失败后，是否可以重做一遍达成恢复目的。
  -例如：插入一条记录失败，再次插入成功。
  -物理日志满足失败可重做性；逻辑日志不满足：例如插入数据页面成功，而插入索引失败，重做插入这个逻辑日志失败。
- 操作可逆性：逆向执行日志记录的操作，可以恢复原来状态（未执行这批操作时的状态）
  -例如第10个页面第100偏移量的值由20改成21，逆操作由21改成20
  -物理日志不可逆（页面偏移量位置可能被后续记录修改），逻辑日志可逆。