

本笔记为阿里云天池龙珠计划SQL训练营的学习内容，链接  
为：<https://tianchi.aliyun.com/specials/promotion/aicampsq>

# Task04：集合运算 - 表的加减法和 join 等

- Task04：集合运算 - 表的加减法和 join 等
  - 4.1 表的加减法
    - 4.1.1 什么是集合运算
    - 4.1.2 表的加法-UNION
      - 4.1.2.1 UNION
      - 4.1.2.2 UNION 与 OR 谓词
      - 4.1.2.3 包含重复行的集合运算 UNION ALL
      - 4.1.2.4 [扩展阅读]bag 模型与 set 模型
      - 4.1.2.5 隐式类型转换
    - 4.1.3 MySQL 8.0 不支持交运算 INTERSECT
      - 4.1.3.1 [扩展阅读]bag 的交运算
    - 4.1.4 差集, 补集与表的减法
      - 4.1.4.1 MySQL 8.0 还不支持 EXCEPT 运算
      - 4.1.4.2 EXCEPT 与 NOT 谓词
      - 4.1.4.3 EXCEPT ALL 与 bag 的差
      - 4.1.4.4 INTERSECT 与 AND 谓词
    - 4.1.5 对称差
      - 4.1.5.1 借助并集和差集迂回实现交集运算 INTERSECT
  - 4.2 连结(JOIN)
    - 4.2.1 内连结(INNER JOIN)
      - 4.2.1.1 使用内连结从两个表获取信息
      - 4.2.1.2 结合 WHERE 子句使用内连结
      - 4.2.1.3 结合 GROUP BY 子句使用内连结
      - 4.2.1.4 自连结(SELF JOIN)
      - 4.2.1.5 内连结与关联子查询
      - 4.2.1.6 自然连结(NATURAL JOIN)
      - 4.2.1.7 使用连结求交集
    - 4.2.2 外连结(OUTER JOIN)
      - 4.2.2.1 左连结与右连接
      - 4.2.2.2 使用左连结从两个表获取信息

- 4.2.2.3 结合 WHERE 子句使用左连结
  - 4.2.2.4 在 MySQL 中实现全外连结
  - 4.2.3 多表连结
    - 4.2.3.1 多表进行内连结
    - 4.2.3.2 多表进行外连结
  - 4.2.4 ON 子句进阶—非等值连结
    - 4.2.4.1 非等值自左连结(SELF JOIN)
  - 4.2.5 交叉连结——CROSS JOIN(笛卡尔积)
    - 4.2.5.1 [扩展阅读]连结与笛卡儿积的关系
  - 4.2.6 连结的特定语法和过时语法
- 练习题
    - 练习题 4.1
    - 练习题 4.2
    - 练习题 4.3
    - 练习题 4.4
    - 练习题 4.5

## 4.1 表的加减法

---

### 4.1.1 什么是集合运算

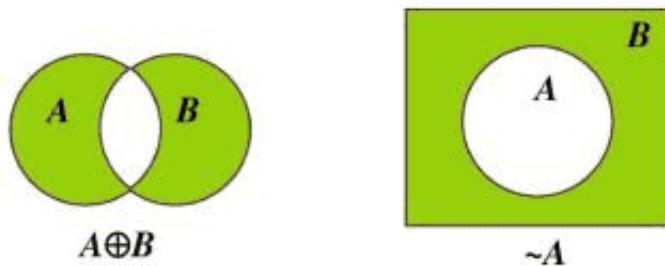
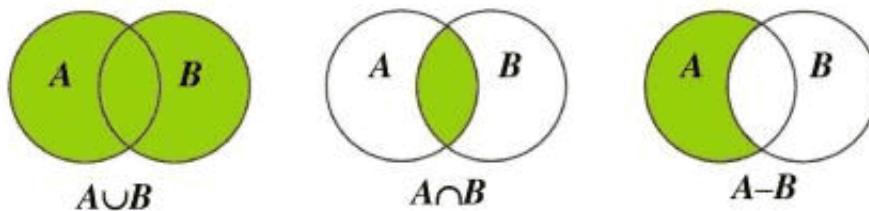
集合 在数学领域表示“各种各样的事物的总和”，在数据库领域表示记录的集合。具体来说，表、视图和查询的执行结果都是记录的集合，其中的元素为表或者查询结果中的每一行。

在标准 SQL 中，分别对检索结果使用 UNION, INTERSECT, EXCEPT 来将检索结果进行并，交和差运算，像 UNION, INTERSECT, EXCEPT 这种用来进行集合运算的运算符称为集合运算符。

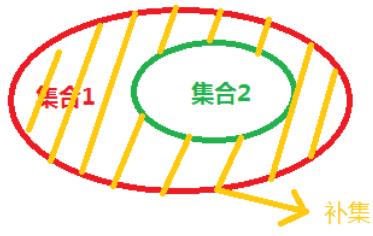
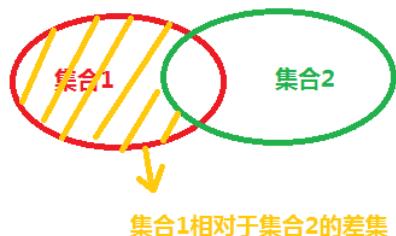
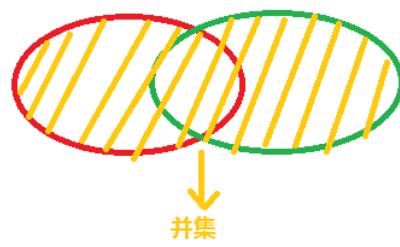
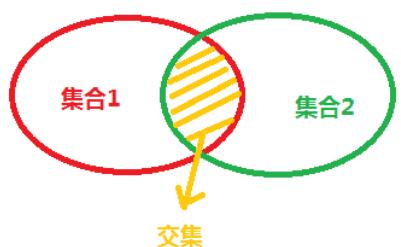
以下的文氏图展示了几种集合的基本运算。

# 文氏图

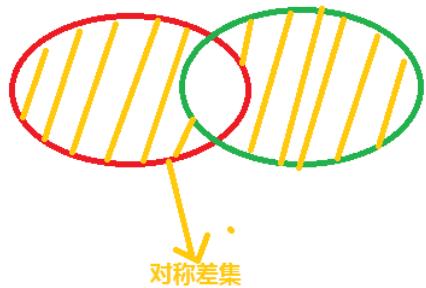
集合运算的表示



7



集合1是集合2的超集  
集合2是集合1的子集



[图片来源于网络]

在数据库中,所有的表-以及查询结果-都可以视为集合,因此也可以把表视为集合进行上述集合运算,很多时候,这种抽象非常有助于对复杂查询问题给出一个可行的思路.

## 4.1.2 表的加法-UNION

### 4.1.2.1 UNION

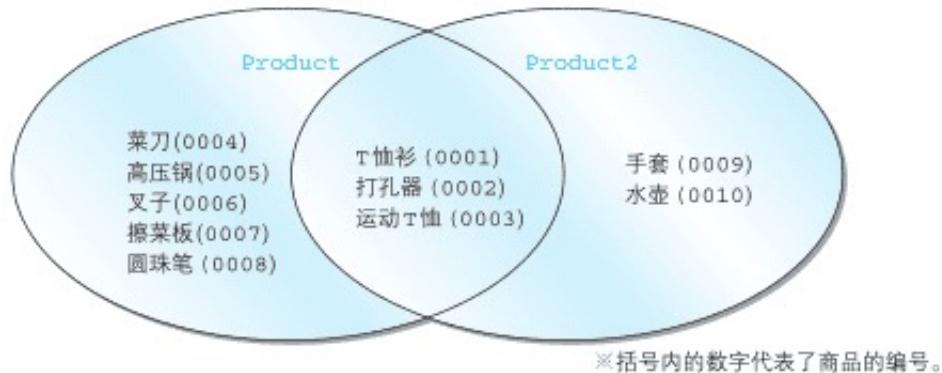
建表代码及数据导入请使用第一章提供的代码.

接下来我们演示 UNION 的具体用法及查询结果:

```
1 | SELECT product_id, product_name
2 | FROM product
3 | UNION
4 | SELECT product_id, product_name
5 | FROM product2;
```

上述结果包含了两张表中的全部商品.你会发现,这就是我们在学校学过的集合中的并集运算,通过文氏图会看得更清晰(图 7-1):

图 7-1 使用 UNION 对表进行加法(并集)运算的图示



通过观察可以发现,商品编号为“0001”~“0003”的3条记录在两个表中都存在,因此大家可能会认为结果中会出现重复的记录,但是 UNION 等集合运算符通常都会除去重复的记录.

上述查询是对不同的两张表进行求并集运算.对于同一张表,实际上也是可以进行求并集的.

**练习题:**假设连锁店想要增加毛利率超过 50% 或者售价低于 800 的货物的存货量,请使用 UNION 对分别满足上述两个条件的商品的查询结果求并集.

结果应该类似于:

product_id	product_name	product_type	sale_price	purchase_price
0002	打孔器	办公用品	500	320
0006	叉子	厨房用具	500	(NULL)
0008	圆珠笔	办公用品	100	(NULL)
0001	T恤	衣服	1000	500

-- 参考答案：  
1    SELECT product\_id, product\_name, product\_type  
2       , sale\_price, purchase\_price

3    FROM product  
4    WHERE sale\_price < 800

5    UNION A

6    SELECT product\_id, product\_name, product\_type  
7       , sale\_price, purchase\_price

8    FROM product  
9    WHERE sale\_price > 1.5 \* purchase\_price;

A or B

思考：如果不使用 UNION 该怎么写查询语句？

-- 参考答案：  
1    SELECT product\_id, product\_name, product\_type  
2       , sale\_price, purchase\_price  
3    FROM product  
4    WHERE sale\_price < 800  
5       OR sale\_price > 1.5 \* purchase\_price;

#### 4.1.2.2 UNION 与 OR 谓词

对于上边的练习题，如果你已经正确地写出来查询，你会发现，使用 UNION 对两个查询结果取并集，和在一个查询中使用 WHERE 子句，然后使用 OR 谓词连接两个查询条件，能够得到相同的结果。

那么是不是就没必要引入 UNION 了呢？当然不是这样的。确实，对于同一个表的两个不同的筛选结果集，使用 UNION 对两个结果集取并集，和把两个子查询的筛选条件用 OR 谓词连接，会得到相同的结果，但倘若要将两个不同的表中的结果合并在一起，就不得不使用 UNION 了。

而且，即便是对于同一张表，有时也会出于查询效率方面的因素来使用 UNION.

练习题：

分别使用 UNION 或者 OR 谓词，找出毛利率不足 30% 或毛利率未知的商品。

参考答案：

```
1 -- 使用 OR 谓词
2 SELECT *
3 FROM product
4 WHERE sale_price / purchase_price < 1.3
5     OR sale_price / purchase_price IS NULL;
```

```
1 -- 使用 UNION
2 SELECT *
3 FROM product
4 WHERE sale_price / purchase_price < 1.3
5
6 UNION
7
8 SELECT *
9 FROM product
10 WHERE sale_price / purchase_price IS NULL;
```

#### 4.1.2.3 包含重复行的集合运算 UNION ALL

在 1.1.1 中我们发现, SQL 语句的 UNION 会对两个查询的结果集进行合并和去重, 这种去重不仅会去掉两个结果集相互重复的, 还会去掉一个结果集中的重复行. 但在实践中有时候需要不去重的并集, 在 UNION 的结果中保留重复行的语法其实非常简单, 只需要在 UNION 后面添加 ALL 关键字就可以了.

例如, 想要知道 product 和 product2 中所包含的商品种类及每种商品的数量, 第一步, 就需要将两个表的商品种类字段选出来, 然后使用 UNION ALL 进行不去重地合并. 接下来再对两个表的结果按 product\_type 字段分组计数.

```
1 -- 保留重复行
2 SELECT product_id, product_name
3 FROM product
4
5 UNION ALL
6
7 SELECT product_id, product_name
8 FROM product2;
```

查询结果如下:

product_id	product_name
0001	T恤
0002	打孔器
0003	运动T恤
0004	菜刀
0005	高压锅
0006	叉子
0007	擦菜板
0008	圆珠笔
0001	T恤
0002	打孔器
0003	运动T恤
0009	手套
0010	水壶

### 练习题:

商店决定对 product 表中利润低于 50% 和售价低于 1000 的商品提价, 请使用 UNION ALL 语句将分别满足上述两个条件的结果取并集. 查询结果类似下表:

product_id	product_name	product_type	sale_price	purchase_price	regist_date
0002	打孔器	办公用品	500	320	2009-09-11
0006	叉子	厨房用具	500	(NULL)	2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100	(NULL)	2009-11-11
0001	T恤	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11

### 参考答案

```

1 | SELECT *
2 | FROM product
3 | WHERE sale_price < 1000
4 |
5 | UNION ALL
6 |
7 | SELECT *
8 | FROM product
9 | WHERE sale_price > 1.5 * purchase_price

```

### 4.1.2.4 [扩展阅读]bag 模型与 set 模型

在高中数学课上我们就学过, 集合的一个显著的特征就是集合中的元素都是互异的. 当我们把数据库中的表看作是集合的时候, 实际上存在一些问题的: 不论是有意的设计或无意的过失, 很多数据库中的表包含了重复的行.

Bag 是和 set 类似的一种数学结构, 不一样的地方在于: bag 里面允许存在重复元素, 如果同

一个元素被加入多次, 则袋子里就有多个该元素.

通过上述 bag 与 set 定义之间的差别我们就发现, 使用 bag 模型来描述数据库中的表很多时候更加合适.

是否允许元素重复导致了 set 和 bag 的并交差等运算都存在一些区别. 以 bag 的交为例, 由于 bag 允许元素重复出现, 对于两个 bag, 他们的并运算会按照: \*\*1. 该元素是否至少在一个 bag 里出现过, 2. 该元素在两个 bag 中的最大出现次数 \*\* 这两个方面来进行计算. 因此对于 A = {1,1,1,2,3,5,7}, B = {1,1,2,2,4,6,8} 两个 bag, 它们的并就等于 {1,1,1,2,2,3,4,5,6,7,8}.

#### 4.1.2.5 隐式类型转换

通常来说, 我们会把类型完全一致, 并且代表相同属性的列使用 UNION 合并到一起显示, 但有时候, 即使数据类型不完全相同, 也会通过隐式类型转换来将两个类型不同的列放在一列里显示, 例如字符串和数值类型:

```
1 | SELECT product_id, product_name, 1'          | a,b      | is      | 月
2 | FROM product                                | sp       |      |      |
3 | UNION
4 | SELECT product_id, product_name,sale_price   |           |      |      |
5 | FROM product2;
```

上述查询能够正确执行, 得到如下结果:

product_id	product_name	1
0001	T恤	1
0002	打孔器	1
0003	运动T恤	1
0004	菜刀	1
0005	高压锅	1
0006	叉子	1
0007	擦菜板	1
0008	圆珠笔	1
0001	T恤	1000
0002	打孔器	500
0003	运动T恤	4000
0009	手套	800
0010	水壶	2000

\*\* 练习题:\*\*

使用 SYSDATE() 函数可以返回当前日期时间, 是一个日期时间类型的数据, 试测试该数据类型和数值, 字符串等类型的兼容性.

例如, 以下代码可以正确执行, 说明时间日期类型和字符串, 数值以及缺失值均能兼容.

```

1 | SELECT SYSDATE(), SYSDATE(), SYSDATE()
2 | _____
3 | UNION
4 |
5 | SELECT 'chars', 123, null

```

上述代码的查询结果:

SYSDATE()	SYSDATE() (1)	SYSDATE() (2)
2020-08-25 15:51:48	2020-08-25 15:51:48	2020-08-25 15:51:48
chars	123	(Null)

### 4.1.3 MySQL 8.0 不支持交运算 INTERSECT

集合的交, 就是两个集合的公共部分, 由于集合元素的互异性, 集合的交只需通过文氏图就可以很直观地看到它的意义.

虽然集合的交运算在 SQL 标准中已经出现多年了, 然而很遗憾的是, 截止到 MySQL 8.0 版本, MySQL 仍然不支持 INTERSECT 操作.

```

1 | SELECT product_id, product_name
2 | FROM product
3 |
4 | INTERSECT
5 |
6 | SELECT product_id, product_name
7 | FROM product2

```

错误代码: 1064

You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'SELECT product\_id,  
product\_name  
FROM product2'

#### 4.1.3.1 [扩展阅读]bag 的交运算

对于两个 bag, 他们的交运算会按照:

1. 该元素是否同时属于两个 bag,
2. 该元素在两个 bag 中的最小出现次数

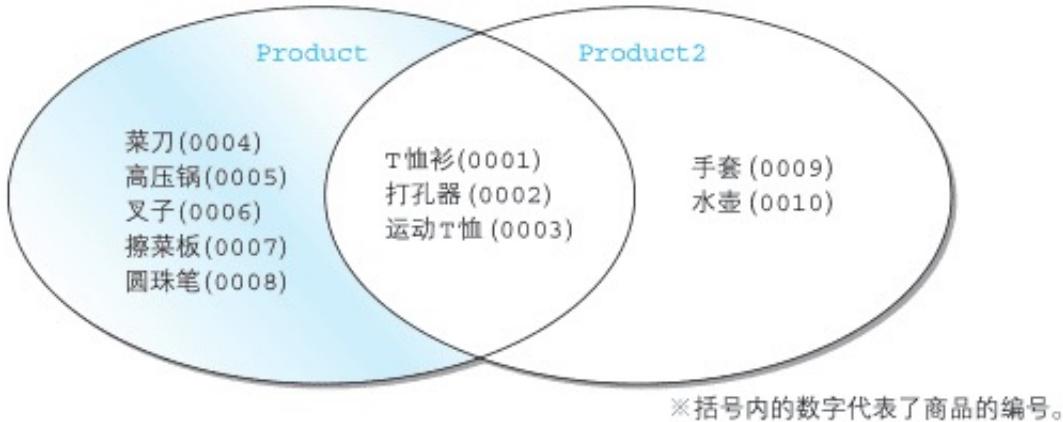
这两个方面来进行计算. 因此对于

A = {1,1,1,2,3,5,7}, B = {1,1,2,2,4,6,8} 两个 bag,  
它们的交运算结果就等于 {1,1,2}.

## 4.1.4 差集, 补集与表的减法

求集合差集的减法运算和实数的减法运算有些不同, 当使用一个集合 A 减去另一个集合 B 的时候, 对于只存在于集合 B 而不存在于集合 A 的元素, 采取直接忽略的策略, 因此集合 A 和 B 做减法只是将集合 A 中也同时属于集合 B 的元素减掉。

图 7-3 使用 EXCEPT 对记录进行减法运算的图示



### 4.1.4.1 MySQL 8.0 还不支持 EXCEPT 运算

MySQL 8.0 还不支持 表的减法运算符 EXCEPT. 不过, 借助第六章学过的 NOT IN 谓词, 我们同样可以实现表的减法.

练习题：

找出只存在于 product 表但不存在于 product2 表的商品.

```
1 | -- 使用 IN 子句的实现方法
2 | SELECT *
3 | FROM product
4 | WHERE product_id NOT IN (SELECT product_id
5 |                           FROM product2)
```

### 4.1.4.2 EXCEPT 与 NOT 谓词

通过上述练习题的 MySQL 解法, 我们发现, 使用 NOT IN 谓词, 基本上可以实现和 SQL 标准语法中的 EXCEPT 运算相同的效果.

练习题:

使用 NOT 谓词进行集合的减法运算, 求出 product 表中, 售价高于 2000, 但利润低于 30% 的商品, 结果应该如下表所示.

product_id	product_name	product_type	sale_price	purchase_price	regist_date
0003	运动T恤	衣服	4000	2800	(NULL)
0005	高压锅	厨房用具	6800	5000	2009-01-15

参考答案:

```

1 | SELECT *
2 | FROM product
3 | WHERE sale_price > 2000
4 | AND product_id NOT IN (
5 |   SELECT product_id
6 |   FROM product
7 |   WHERE sale_price < 1.3*purchase_price)

```

#### 4.1.4.3 EXCEPT ALL 与 bag 的差

类似于 UNION ALL, EXCEPT ALL 也是按出现次数进行减法, 也是使用 bag 模型进行运算.

对于两个 bag, 他们的差运算会按照:

1. 该元素是否属于作为被减数的 bag,
2. 该元素在两个 bag 中的出现次数

这两个方面来进行计算. 只有属于被减数的 bag 的元素才参与 EXCEPT ALL 运算, 并且差 bag 中的次数, 等于该元素在两个 bag 的出现次数之差(差为零或负数则不出现). 因此对于 A = {1,1,1,2,3,5,7}, B = {1,1,2,2,4,6,8} 两个 bag, 它们的差就等于 {1,3,5,7}.

#### 4.1.4.4 INTERSECT 与 AND 谓词

对于同一个表的两个查询结果而言, 他们的交 INTERSECT 实际上可以等价地将两个查询的检索条件用 AND 谓词连接来实现.

练习题:

使用 AND 谓词查找 product 表中利润率高于 50%, 并且售价低于 1500 的商品, 查询结果如下所示.

product_id	product_name	product_type	sale_price	purchase_price	regist_date
0001	T恤	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11

参考答案

```
| SELECT *
```

```
1 | FROM product
2 | WHERE sale_price > 1.5 * purchase_price
3 |     AND sale_price < 1500
4 |
```

#### 4.1.5 对称差

两个集合 A,B 的对称差是指那些仅属于 A 或仅属于 B 的元素构成的集合. 对称差也是个非常基础的运算, 例如, 两个集合的交就可以看作是两个集合的并去掉两个集合的对称差. 上述方法在其他数据库里也可以用来简单地实现表或查询结果的对称差运算: 首先使用 UNION 求两个表的并集, 然后使用 INTERSECT 求两个表的交集, 然后用并集减去交集, 就得到了对称差.

但由于在 MySQL 8.0 里, 由于两个表或查询结果的并不能直接求出来, 因此并不适合使用上述思路来求对称差. 好在还有差集运算可以使用. 从直观上就能看出来, 两个集合的对称差等于 A-B 并上 B-A, 因此实践中可以用这个思路来求对称差.

练习题:

使用 product 表和 product2 表的对称差来查询哪些商品只在其中一张表, 结果类似于:

product_id	product_name	product_type	sale_price	purchase_price	regist_date
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500	(NULL)	2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100	(NULL)	2009-11-11
0009	手套	衣服	800	500	(NULL)
0010	水壶	厨房用具	2000	1700	2009-09-20

提示: 使用 NOT IN 实现两个表的差集.

参考答案:

```
1 | -- 使用 NOT IN 实现两个表的差集
2 | SELECT *
3 | FROM product
4 | WHERE product_id NOT IN (
5 |     SELECT product_id FROM product2)
6 |
7 | UNION
8 |
9 | SELECT *
10 | FROM product2
11 | WHERE product_id NOT IN (
12 |     SELECT product_id FROM product)
```

#### 4.1.5.1 借助并集和差集迂回实现交集运算 INTERSECT

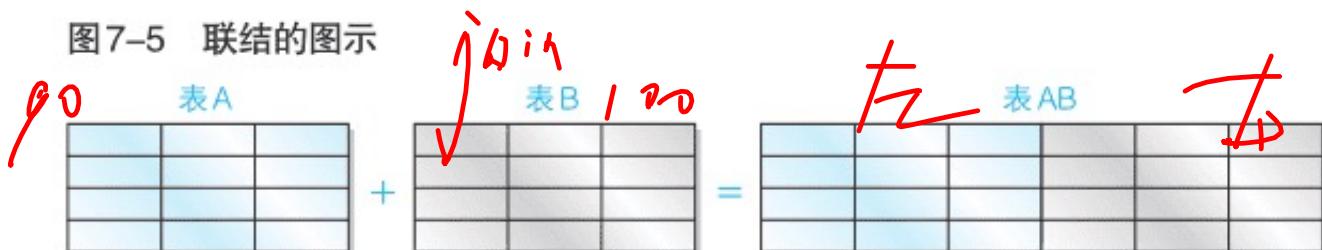
通过观察集合运算的文氏图, 我们发现, 两个集合的交可以看作是两个集合的并去掉两个集合的对称差。

## 4.2 连结(JOIN)

前一节我们学习了 UNION 和 INTERSECT 等集合运算, 这些集合运算的特征就是以行方向为单位进行操作. 通俗地说, 就是进行这些集合运算时, 会导致记录行数的增减. 使用 UNION 会增加记录行数, 而使用 INTERSECT 或者 EXCEPT 会减少记录行数.

但这些运算不能改变列的变化, 虽然使用函数或者 CASE 表达式等列运算, 可以增加列的数量, 但仍然只能从一张表中提供的基础信息列中获得一些 "引申列", 本质上并不能提供更多的信息. 如果想要从多个表获取信息, 例如, 如果我们想要找出某个商店里的衣服类商品的名称, 数量及价格等信息, 则必须分别从 shopproduct 表和 product 表获取信息.

图 7-5 联结的图示



注:

截至目前, 本书中出现的示例(除了关联子查询)基本上都是从一张表中选取数据, 但实际上, 期望得到的数据往往会分散在不同的表之中, 这时候就需要使用连结了.

之前在学习关联子查询时我们发现, 使用关联子查询也可以从其他表获取信息, 但 \*\* 连结 \*\* 更适合从多张表获取信息.

连结 (JOIN) 就是使用某种关联条件(一般是使用相等判断谓词 "="), 将其他表中的列添加过来, 进行“添加列”的集合运算. 可以说, 连结是 SQL 查询的核心操作, 掌握了连结, 能够从两张甚至多张表中获取列, 能够将过去使用关联子查询等过于复杂的查询简化为更加易读的形式, 以及进行一些更加复杂的查询.

SQL 中的连结有多种分类方法, 我们这里使用最基础的内连结和外连结的分类方法来分别进行讲解.

### 4.2.1 内连结(INNER JOIN)



内连结的语法格式是:



```
2 | FROM <tb_1> INNER JOIN <tb_2> ON <condition(s)>
```

其中 INNER 关键词表示使用了内连结, 至于内连结的涵义, 目前暂时可以不必细究. 例如, 还是刚才那个问题:

找出某个商店里的衣服类商品的名称, 数量及价格等信息.

我们进一步把这个问题明确化:

找出东京商店里的衣服类商品的商品名称, 商品价格, 商品种类, 商品数量信息.

#### 4.2.1.1 使用内连结从两个表获取信息

我们先来分别观察所涉及的表, product 表保存了商品编号, 商品名称, 商品种类等信息, 这个表可以提供关于衣服种类的衣服的详细信息, 但是不能提供商店信息.

表 7-1 Product(商品)表

product_id (商品编号)	product_name (商品名称)	product_type (商品种类)	sale_price (销售单价)	purchase_price (进货单价)	regist_date (登记日期)
0001	T恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动T恤	衣服	4000	2800	
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500		2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100		2009-11-11

我们接下来观察 shopproduct 表, 这个表里有商店编号名称, 商店的商品编号及数量. 但要想获取商品的种类及名称售价等信息, 则必须借助于 product 表.

表7-2 ShopProduct(商店商品)表

shop_id (商店编号)	shop_name (商店名称)	product_id (商品编号)	quantity (数量)
000A	东京	0001	30
000A	东京	0002	50
000A	东京	0003	15
000B	名古屋	0002	30
000B	名古屋	0003	120
000B	名古屋	0004	20
000B	名古屋	0006	10
000B	名古屋	0007	40
000C	大阪	0003	20
000C	大阪	0004	50

所以问题的关键是, 找出一个类似于 "轴" 或者 "桥梁" 的公共列, 将两张表用这个列连结起来. 这就是连结运算所要作的事情.

我们来对比一下上述两张表, 可以发现, 商品编号列是一个公共列, 因此很自然的事情就是用这个商品编号列来作为连接的“桥梁”, 将 **product** 和 **shopproduct** 这两张表连接起来。

表7-2 ShopProduct(商店商品)表

shop_id (商店编号)	shop_name (商店名称)	product_id (商品编号)	quantity (数量)
000A	东京	0001	30
000A	东京	0002	50
000A	东京	0003	15
000B	名古屋	0002	30
000B	名古屋	0003	120
000B	名古屋	0004	20
000B	名古屋	0006	10
000B	名古屋	0007	40
000C	大阪	0003	20
000C	大阪	0004	50

注:

如果你使用过 excel 的 vlookup 函数, 你会发现这个函数正好也能够实现这个功能. 实际上, 在思路上, 关联子查询更像是 vlookup 函数: 以表 A 为主表, 然后根据表 A 的关联列的每一行的取值, 逐个到表 B 中的关联列中去查找取值相等的行.

当数据量较少时, 这种方式并不会有什么性能问题, 但数据量较大时, 这种方式将会导致较大的计算开销: 对于外部查询返回的每一行数据, 都会向内部的子查询传递一个关联列的值, 然后内部子查询根据传入的值执行一次查询然后返回它的查询结果. 这就使得, 例如外部主查询的返回结果有一万行, 那么子查询就会执行一万次, 这将会带来非常恐怖的时间消耗.

我们把上述问题进行分解:

首先, 找出每个商店的商店编号, 商店名称, 商品编号, 商品名称, 商品类别, 商品售价, 商品数量信息.

按照内连结的语法, 在 FROM 子句中使用 INNER JOIN 将两张表连接起来, 并为 ON 子句指定连结条件为 `shop_product.product_id=product.product_id`, 就得到了如下的查询语句:

```
1 | SELECT SP.shop_id
2 |   ,SP.shop_name
3 |   ,SP.product_id
4 |   ,P.product_name
5 |   ,P.product_type
6 |   ,P.sale_price
7 |   ,SP.quantity
8 | FROM shop_product AS SP
9 | INNER JOIN product AS P
10| ON SP.product_id = P.product_id;
```

在上述查询中, 我们分别为两张表指定了简单的别名, 这种操作在使用连结时是非常常见的, 通过别名会让我们在编写查询时少打很多字, 并且更重要的是, 会让查询语句看起来更加简洁.

上述查询将会得到如下的结果:

shop_id	shop_name	product_id	product_name	product_type	sale_price	quantity
000A	东京	0001	T恤	衣服	1000	30
000A	东京	0002	打孔器	办公用品	500	50
000A	东京	0003	运动T恤	衣服	4000	15
000B	名古屋	0002	打孔器	办公用品	500	30
000B	名古屋	0003	运动T恤	衣服	4000	120
000B	名古屋	0004	菜刀	厨房用具	3000	20
000B	名古屋	0006	叉子	厨房用具	500	10
000B	名古屋	0007	擦菜板	厨房用具	880	40
000C	大阪	0003	运动T恤	衣服	4000	20
000C	大阪	0004	菜刀	厨房用具	3000	50
000C	大阪	0006	叉子	厨房用具	500	90
000C	大阪	0007	擦菜板	厨房用具	880	70
000D	福冈	0001	T恤	衣服	1000	100

观察查询结果, 我们看到, 这个结果里的列已经包含了所有我们需要的信息.

关于内连结, 需要注意以下三点:

### 要点一: 进行连结时需要在 FROM 子句中使用多张表.

之前的 FROM 子句中只有一张表, 而这次我们同时使用了 shopproduct 和 product 两张表, 使用关键字 INNER JOIN 就可以将两张表连结在一起了:

```
1 | FROM shop_product AS SP INNER JOIN product AS P
```

### 要点二: 必须使用 ON 子句来指定连结条件.

在进行内连结时 ON 子句是必不可少的(大家可以试试去掉上述查询的 ON 子句后会有什么结果).

ON 子句是专门用来指定连结条件的, 我们在上述查询的 ON 之后指定两张表连结所使用的列以及比较条件, 基本上, 它能起到与 WHERE 相同的筛选作用, 我们会在本章的结尾部分进一步探讨这个话题.

### 要点三: SELECT 子句中的列最好按照 表名.列名 的格式来使用.

当两张表的列除了用于关联的列之外, 没有名称相同的列的时候, 也可以不写表名, 但表名使得我们能够在今后的任何时间阅读查询代码的时候, 都能马上看出每一列来自于哪张表, 能够节省我们很多时间.

但是, 如果两张表有其他名称相同的列, 则必须使用上述格式来选择列名, 否则查询语句会报错.

我们回到上述查询所回答的问题. 通过观察上述查询的结果, 我们发现, 这个结果离我们的目标: 找出东京商店的衣服类商品的基础信息已经很接近了. 接下来, 我们只需要把这个查询结果作为一张表, 给它增加一个 WHERE 子句来指定筛选条件.

#### 4.2.1.2 结合 WHERE 子句使用内连结

如果需要在使用内连结的时候同时使用 WHERE 子句对检索结果进行筛选, 则需要把 WHERE 子句写在 ON 子句的后边.

例如, 对于上述查询问题, 我们可以在前一步查询的基础上, 增加 WHERE 条件.

增加 WHERE 子句的方式有好几种, 我们先从最简单的说起.

第一种增加 WHERE 子句的方式, 就是把上述查询作为子查询, 用括号封装起来, 然后在外层查询增加筛选条件.

```
1 | SELECT *
2 | FROM ( -- 第一步查询的结果
3 |         SELECT SP.shop_id
4 |                 ,SP.shop_name
5 |                 ,SP.product_id
6 |                 ,P.product_name
7 |                 ,P.product_type
8 |                 ,P.sale_price
9 |                 ,SP.quantity
10 |            FROM shop_product AS SP
11 |            INNER JOIN product AS P
12 |                  ON SP.product_id = P.product_id) AS STEP1
13 | WHERE shop_name = '东京'
14 |       AND product_type = '衣服' ;
```

还记得我们学习子查询时的认识吗? 子查询的结果其实也是一张表, 只不过是一张虚拟的表, 它并不真实存在于数据库中, 只是数据库中其他表经过筛选, 聚合等查询操作后得到的一个"视图".

这种写法能很清晰地分辨出每一个操作步骤, 在我们还不十分熟悉 SQL 查询每一个子句的执行顺序的时候可以帮到我们.

但实际上, 如果我们熟知 WHERE 子句将在 FROM 子句之后执行, 也就是说, 在做完 INNER JOIN ... ON 得到一个新表后, 才会执行 WHERE 子句, 那么就得到标准的写法:

```
1 | SELECT SP.shop_id
2 |         ,SP.shop_name
3 |         ,SP.product_id
4 |         ,P.product_name
5 |         ,P.product_type
6 |         ,P.sale_price
7 |         ,SP.quantity
8 |    FROM shopproduct AS SP
9 |    INNER JOIN product AS P
10 |           ON SP.product_id = P.product_id
```

```

11 | WHERE SP.shop_name = '东京'
12 |     AND P.product_type = '衣服' ;

```

我们首先给出上述查询的执行顺序:

### FROM 子句 ->WHERE 子句 ->SELECT 子句

也就是说, 两张表是先按照连结列进行了连结, 得到了一张新表, 然后 WHERE 子句对这张新表的行按照两个条件进行了筛选, 最后, SELECT 子句选出了那些我们需要的列.

此外, 一种不是很常见的做法是, 还可以将 WHERE 子句中的条件直接添加在 ON 子句中, 这时候 ON 子句后最好用括号将连结条件和筛选条件括起来.

```

1 | SELECT SP.shop_id
2 |     ,SP.shop_name
3 |     ,SP.product_id
4 |     ,P.product_name
5 |     ,P.product_type
6 |     ,P.sale_price
7 |     ,SP.quantity
8 | FROM shop_product AS SP
9 | INNER JOIN product AS P
10 |    ON (SP.product_id = P.product_id
11 |        AND SP.shop_name = '东京'
12 |        AND P.product_type = '衣服') ;

```

```

SELECT SP.shop_id
,SP.shop_name
,SP.product_id
,P.product_name
,P.product_type
,P.sale_price
,SP.quantity
FROM shop_product AS SP
INNER JOIN product AS P
ON (SP.product_id = P.product_id
AND SP.shop_name = '东京'
AND P.product_type = '衣服') ;

```

1. 把 0005 排除了
2. 把 4, 6, 7, 8 排除
3. 把 2 排除了

product_id	product_name	product_type	sale_price	purchase_price	regist_date
0001	T恤衫	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动T恤	衣服	4000	2800	NULL
0004	菜刀	厨房用具	3000	2800	2009-09-20
0005	高压锅	厨房用具	6800	5000	2009-01-15
0006	叉子	厨房用具	500	NULL	2009-09-20
0007	擦菜板	厨房用具	880	790	2008-04-28
0008	圆珠笔	办公用品	100	NULL	2009-11-11

shop_id	shop_name	product_id	quantity
000A	东京	0001	30
000A	东京	0002	50
000A	东京	0003	15
000B	名古屋	0002	30
000B	名古屋	0003	120
000B	名古屋	0004	20
000B	名古屋	0006	10
000B	名古屋	0007	40
000C	大阪	0003	20
000C	大阪	0004	50
000C	大阪	0006	90
000C	大阪	0007	70
000D	福冈	0001	100

但上述这种把筛选条件和连结条件都放在 ON 子句的写法, 不是太容易阅读, 不建议大家使用.

另外, 先连结再筛选的标准写法的执行顺序是, 两张完整的表做了连结之后再做筛选, 如果要连结多张表, 或者需要做的筛选比较复杂时, 在写 SQL 查询时会感觉比较吃力. 在结合 WHERE 子句使用内连结的时候, 我们也可以更改任务顺序, 并采用任务分解的方法, 先分别

在两个表使用 WHERE 进行筛选，然后把上述两个子查询连结起来。

```
1 | SELECT SP.shop_id
2 |   ,SP.shop_name
3 |   ,SP.product_id
4 |   ,P.product_name
5 |   ,P.product_type
6 |   ,P.sale_price
7 |   ,SP.quantity
8 | FROM (-- 子查询 1: 从 shopproduct 表筛选出东京商店的信息
9 |       SELECT *
10 |         FROM shop_product
11 |        WHERE shop_name = '东京' ) AS SP
12 | INNER JOIN -- 子查询 2: 从 product 表筛选出衣服类商品的信息
13 |   (SELECT *
14 |     FROM product
15 |    WHERE product_type = '衣服') AS P
16 |
17 |   ON SP.product_id = P.product_id;
```

序号	shop_id	shop_name	product_id	product_name	product_type	sale_price	quantity
1	000A	东京	0001	运动T恤	衣服	4,000	30
2	000A	东京	0001	T恤衫	衣服	1,000	30
3	000A	东京	0002	运动T恤	衣服	4,000	50
4	000A	东京	0002	T恤衫	衣服	1,000	50
5	000A	东京	0003	运动T恤	衣服	4,000	15
6	000A	东京	0003	T恤衫	衣服	1,000	15

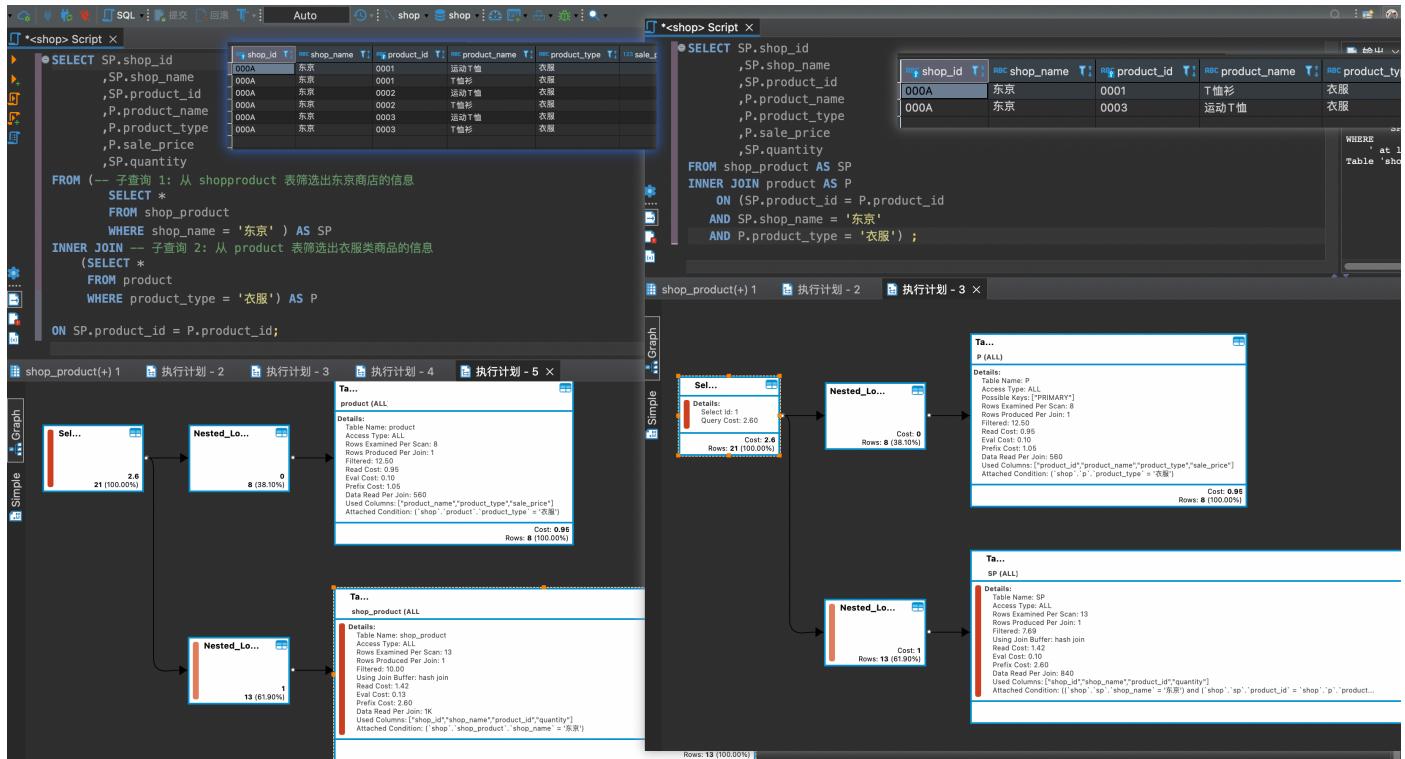
1. 把名，大，福去掉  
2. 把办，厨去掉  
3. ON SP.product\_id = P.product\_id;

1. 把名，大，福去掉  
2. 把办，厨去掉

shop_id	shop_name	product_id	quantity
000A	东京	0001	30
000A	东京	0002	50
000A	东京	0003	15
000B	名古屋	0002	30
000B	名古屋	0003	120
000B	名古屋	0004	20
000B	名古屋	0006	10
000B	名古屋	0007	40
000C	大阪	0003	20
000C	大阪	0004	50
000C	大阪	0006	90
000C	大阪	0007	70
000D	福冈	0001	100

先分别在两张表里做筛选，把复杂的筛选条件按表分拆，然后把筛选结果（作为表）连接起来，

避免了写复杂的筛选条件,因此这种看似复杂的写法,实际上整体的逻辑反而非常清晰.在写查询的过程中,首先要按照最便于自己理解的方式来写,先把问题解决了,再思考优化的问题.



## 练习题:

找出每个商店里的衣服类商品的名称及价格等信息. 希望得到如下结果:

shop_id	shop_name	product_id	product_name	product_type	purchase_price
000A	东京	0001	T恤	衣服	500
000A	东京	0003	运动T恤	衣服	2800
000B	名古屋	0003	运动T恤	衣服	2800
000C	大阪	0003	运动T恤	衣服	2800
000D	福冈	0001	T恤	衣服	500

```

1 -- 参考答案 1-- 不使用子查询
2 SELECT SP.shop_id,
3        SP.shop_name,
4        SP.product_id,
5        P.product_name,
6        P.product_type,
7        P.purchase_price
8 FROM shopproduct AS SP
9
10 INNER JOIN product AS P
11
12 ON SP.product_id = P.product_id
13
14 WHERE P.product_type = '衣服';

```

```

1 -- 参考答案 2-- 使用子查询
2
3 SELECT
4     SP.shop_id,
5     SP.shop_name,
6     SP.product_id,
7     P.product_name,
8     P.product_type,
9     P.purchase_price
10    FROM shopproduct AS SP
11
12 INNER JOIN -- 从 product 表找出衣服类商品的信息
13    (SELECT
14        product_id,
15        product_name,
16        product_type,
17        purchase_price
18        FROM product
19        WHERE product_type = '衣服') AS P
20
21    ON SP.product_id = P.product_id;

```

上述第二种写法虽然包含了子查询，并且代码行数更多，但由于每一层的目的很明确，更适于阅读，并且在外连结的情形下，还能避免错误使用 WHERE 子句导致外连结失效的问题，相关示例见后文中的 "结合 WHERE 子句使用外连结" 章节。

### 练习题：

分别使用连结两个子查询和不使用子查询的方式，找出东京商店里，售价低于 2000 的商品信息，希望得到如下结果。

shop_id	shop_name	product_id	quantity	product_id(1)	product_name	product_type	sale_price
000A	东京	0001	30	0001	T恤	衣服	1000
000A	东京	0002	50	0002	打孔器	办公用品	500

```

1 -- 参考答案
2 -- 不使用子查询
3 SELECT SP.* , P. *
4 FROM shopproduct AS SP
5
6 INNER JOIN product AS P
7
8 ON SP.product_id = P.product_id
9
10 WHERE shop_id = '000A' AND sale_price < 2000;

```

#### 4.2.1.3 结合 GROUP BY 子句使用内连结

结合 GROUP BY 子句使用内连结, 需要根据分组列位于哪个表区别对待.

最简单的情形, 是在内连结之前就使用 GROUP BY 子句.

但是如果分组列和被聚合的列不在同一张表, 且二者都未被用于连结两张表, 则只能先连结, 再聚合.

练习题:

每个商店中, 售价最高的商品的售价分别是多少?

```
1 -- 参考答案
2 SELECT SP.shop_id
3     ,SP.shop_name
4     ,MAX(P.sale_price) AS max_price
5 FROM shopproduct AS SP
6
7 INNER JOIN product AS P
8
9 ON SP.product_id = P.product_id
10
11 GROUP BY SP.shop_id,SP.shop_name
```

**思考题:** 上述查询得到了每个商品售价最高的商品, 但并不知道售价最高的商品是哪一个. 如何获取每个商店里售价最高的商品的名称和售价?

注: 这道题的一个简易的方式是使用下一章的窗口函数. 当然, 也可以使用其他我们已经学过的知识来实现, 例如, 在找出每个商店售价最高商品的价格后, 使用这个价格再与 product 列进行连结, 但这种做法在价格不唯一时会出现问题.

#### 4.2.1.4 自连结(SELF JOIN)

之前的内连结, 连结的都是不一样的两个表. 但实际上一张表也可以与自身作连结, 这种连接称之为自连结. 需要注意, 自连结并不是区分于内连结和外连结的第三种连结, 自连结可以是外连结也可以是内连结, 它是不同于内连结外连结的另一个连结的分类方法.

#### 4.2.1.5 内连结与关联子查询

回忆第五章第三节关联子查询中的问题: 找出每个商品种类当中售价高于该类商品的平均售价的商品. 当时我们是使用关联子查询来实现的.

```
1 | SELECT
2 |     product_type,
```

```
3     product_name,  
4     sale_price  
5 FROM product AS P1  
6  
7 WHERE sale_price > (SELECT AVG(sale_price)  
8                 FROM product AS P2  
9                 WHERE P1.product_type = P2.product_type  
10                GROUP BY product_type);
```

使用内连结同样可以解决这个问题:

首先, 使用 GROUP BY 按商品类别分类计算每类商品的平均价格.

```
1 SELECT  
2     product_type,  
3     AVG(sale_price) AS avg_price  
4 FROM product  
5 GROUP BY product_type;
```

接下来, 将上述查询与表 product 按照 product\_type (商品种类)进行内连结.

```
1 SELECT P1.product_id,  
2     P1.product_name,  
3     P1.product_type,  
4     P1.sale_price,  
5     P2.avg_price  
6 FROM product AS P1  
7  
8 INNER JOIN  
9     (SELECT product_type, AVG(sale_price) AS avg_price  
10    FROM product  
11   GROUP BY product_type) AS P2  
12  
13 ON P1.product_type = P2.product_type;
```

最后, 增加 WHERE 子句, 找出那些售价高于该类商品平均价格的商品. 完整的代码如下:

```
1 SELECT P1.product_id,  
2     P1.product_name,  
3     P1.product_type,  
4     P1.sale_price,  
5     P2.avg_price  
6 FROM product AS P1  
7  
8 INNER JOIN (SELECT product_type, AVG(sale_price) AS avg_price  
9                 FROM product  
10                GROUP BY product_type) AS P2  
10
```

```
11  
12 ON P1.product_type = P2.product_type  
13  
14 WHERE P1.sale_price > P2.avg_price;
```

仅仅从代码量上来看, 上述方法似乎比关联子查询更加复杂, 但这并不意味着这些代码更难理解. 通过上述分析, 很容易发现上述代码的逻辑实际上更符合我们的思路, 因此尽管看起来复杂, 但思路实际上更加清晰.

作为对比, 试分析如下代码:

```
1 SELECT  
2     P1.product_id,  
3     P1.product_name,  
4     P1.product_type,  
5     P1.sale_price, AVG(P2.sale_price) AS avg_price  
6 FROM product AS P1  
7  
8 INNER JOIN product AS P2  
9  
10 ON P1.product_type=P2.product_type  
11  
12 WHERE P1.sale_price > P2.sale_price  
13  
14 GROUP BY  
    P1.product_id, P1.product_name, P1.product_type, P1.sale_price, P2.product_type
```

虽然去掉了子查询, 查询语句的层次更少, 而且代码行数似乎更少, 但实际上这个方法可能更加难以写出来. 在实践中, 一定要按照易于让自己理解的思路去分层次写代码, 而不要花费很长世间写出一个效率可能更高但自己和他人理解起来难度更高的代码.

#### 4.2.1.6 自然连结(NATURAL JOIN)

自然连结并不是区别于内连结和外连结的第三种连结, 它其实是内连结的一种特例-当两个表进行自然连结时, 会按照两个表中都包含的列名来进行等值内连结, 此时无需使用 ON 来指定连接条件.

```
1 | SELECT * FROM shopproduct NATURAL JOIN product
```

上述查询得到的结果, 会把两个表的公共列 (这里是 product\_id, 可以有多个公共列) 放在第一列, 然后按照两个表的顺序和表中列的顺序, 将两个表中的其他列都罗列出来.

shop_id	shop_name	product_id	quantity	product_id(1)	product_name	product_type	sale_price	purchase_price	regist_date
000A	东京	0001	30	0001	T恤	衣服	1000	500	2009-09-20
000A	东京	0002	50	0002	打孔器	办公用品	500	320	2009-09-11
000A	东京	0003	15	0003	运动T恤	衣服	4000	2800	(Null)
000B	名古屋	0002	30	0002	打孔器	办公用品	500	320	2009-09-11
000B	名古屋	0003	120	0003	运动T恤	衣服	4000	2800	(Null)
000B	名古屋	0004	20	0004	菜刀	厨房用具	3000	2800	2009-09-20
000B	名古屋	0006	10	0006	叉子	厨房用具	500	(Null)	2009-09-20
000B	名古屋	0007	40	0007	擦菜板	厨房用具	880	790	2008-04-28
000C	大阪	0003	20	0003	运动T恤	衣服	4000	2800	(Null)
000C	大阪	0004	50	0004	菜刀	厨房用具	3000	2800	2009-09-20
000C	大阪	0006	90	0006	叉子	厨房用具	500	(Null)	2009-09-20
000C	大阪	0007	70	0007	擦菜板	厨房用具	880	790	2008-04-28
000D	福冈	0001	100	0001	T恤	衣服	1000	500	2009-09-20

## 练习题:

试写出与上述自然连结等价的内连结.

```

1 -- 参考答案
2 SELECT
3     SP.product_id,
4     SP.shop_id,
5     SP.shop_name,
6     SP.quantity,
7     P.product_name,
8     P.product_type,
9     P.sale_price,
10    P.purchase_price,
11    P.regist_date
12
13 FROM shopproduct AS SP
14
15 INNER JOIN product AS P
16
17 ON SP.product_id = P.product_id

```

使用自然连结还可以求出两张表或子查询的公共部分, 例如教材中 7-1 选取表中公共部分 - INTERSECT 一节中的问题: 求表 product 和表 product2 中的公共部分, 也可以用自然连结来实现:

```
1 | SELECT * FROM product NATURAL JOIN product2
```

product_id	product_name	product_type	sale_price	purchase_price	regist_date
0001	T恤	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11

这个结果和书上给的结果并不一致, 少了运动 T 恤, 这是由于运动 T 恤的 regist\_date 字段为空, 在进行自然连结时, 来自于 product 和 product2 的运动 T 恤这一行数据在进行比较时, 实际上是在逐字段进行等值连结, 回忆我们在 **6.2 ISNULL,IS NOT NULL** 这一节学到的缺失值的比较方法就可得知, 两个缺失值用等号进行比较, 结果不为真. 而连结只会返回对连结条

件返回为真的那些行.

如果我们将查询语句进行修改:

```
1 | SELECT *
2 | FROM (SELECT
3 |         product_id,
4 |         product_name
5 |     FROM product ) AS A
6 |
7 | NATURAL JOIN
8 |
9 | (SELECT
10 |         product_id,
11 |         product_name
12 |     FROM product2) AS B;
```

那就可以得到正确的结果了:

product_id	product_name
0001	T恤
0002	打孔器
0003	运动T恤

#### 4.2.1.7 使用连结求交集

我们在上一节表的加减法里知道, MySQL 8.0 里没有交集运算, 我们当时是通过并集和差集来实现求交集的. 现在学了连结, 让我们试试使用连结来实现求交集的运算.

练习题: 使用内连结求 product 表和 product2 表的交集.

```
1 | SELECT P1.*
2 |
3 | FROM product AS P1
4 |
5 | INNER JOIN product2 AS P2
6 |
7 | ON (P1.product_id = P2.product_id
8 |      AND P1.product_name = P2.product_name
9 |      AND P1.product_type = P2.product_type
10 |     AND P1.sale_price = P2.sale_price
11 |     AND P1.regist_date = P2.regist_date)
```

得到如下结果

product_id	product_name	product_type	sale_price	purchase_price	regist_date
0001	T恤	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11

注意上述结果和 P230 的结果并不一致-少了 product\_id='0001' 这一行, 观察源表数据可发现, 少的这行数据的 regist\_date 为缺失值, 回忆第六章讲到的 IS NULL 谓词, 我们得知, 这是由于缺失值是不能用等号进行比较导致的.

如果我们仅仅用 product\_id 来进行连结:

```

1 | SELECT P1.*
2 |
3 | FROM product AS P1
4 |
5 | INNER JOIN product2 AS P2
6 |     ON P1.product_id = P2.product_id

```

查询结果:

product_id	product_name	product_type	sale_price	purchase_price	regist_date
0001	T恤	衣服	1000	500	2009-09-20
0002	打孔器	办公用品	500	320	2009-09-11
0003	运动T恤	衣服	4000	2800	(Null)

这次就一致了.

## 4.2.2 外连结(OUTER JOIN)

内连结会丢弃两张表中不满足 ON 条件的行, 和内连结相对的就是外连结. 外连结会根据外连结的种类有选择地保留无法匹配到的行.

按照保留的行位于哪张表, 外连结有三种形式: 左连结, 右连结和全外连结 .

左连结会保存左表中无法按照 ON 子句匹配到的行, 此时对应右表的行均为缺失值; 右连结则会保存右表中无法按照 ON 子句匹配到的行, 此时对应左表的行均为缺失值; 而全外连结则会同时保存两个表中无法按照 ON 子句匹配到的行, 相应的另一张表中的行用缺失值填充.

三种外连结的对应语法分别为:

```

1 | -- 左连结
2 | FROM <tb_1> LEFT OUTER JOIN <tb_2> ON <condition(s)>
3 | -- 右连结
4 | FROM <tb_1> RIGHT OUTER JOIN <tb_2> ON <condition(s)>
5 | -- 全外连结
6 | FROM <tb_1> FULL OUTER JOIN <tb_2> ON <condition(s)>

```

#### 4.2.2.1 左连结与右连接

由于连结时可以交换左表和右表的位置, 因此左连结和右连结并没有本质区别. 接下来我们先以左连结为例进行学习. 所有的内容在调换两个表的前后位置, 并将左连结改为右连结之后, 都能得到相同的结果. 稍后再介绍全外连结的概念.

#### 4.2.2.2 使用左连结从两个表获取信息

如果你仔细观察过将 **shopproduct** 和 **product** 进行内连结前后的结果的话, 你就会发现, **product** 表中有两种商品并未在内连结的结果里, 就是说, 这两种商品并未在任何商店有售(这通常意味着比较重要的业务信息, 例如, 这两种商品在所有商店都处于缺货状态, 需要及时补货). 现在, 让我们先把之前内连结的 SELECT 语句转换为左连结试试看吧.

练习题: 统计每种商品分别在哪些商店有售, 需要包括那些在每个商店都没货的商品.

使用左连结的代码如下(注意区别于书上的右连结):

```
1 | SELECT
2 |     SP.shop_id,
3 |     SP.shop_name,
4 |     SP.product_id,
5 |     P.product_name,
6 |     P.sale_price
7 |
8 | FROM product AS P
9 |
10| LEFT OUTER JOIN shopproduct AS SP
11|
12| ON SP.product_id = P.product_id;
13|
```

上述查询得到的检索结果如下(由于并未使用 ORDER BY 子句指定顺序, 你执行上述代码得到的结果可能顺序与下图不同):

shop_id	shop_name	product_id	product_name	sale_price
000A	东京	0001	T恤	1000
000D	福冈	0001	T恤	1000
000A	东京	0002	打孔器	500
000B	名古屋	0002	打孔器	500
000A	东京	0003	运动T恤	4000
000B	名古屋	0003	运动T恤	4000
000C	大阪	0003	运动T恤	4000
000B	名古屋	0004	菜刀	3000
000C	大阪	0004	菜刀	3000
(NULL)	(NULL)	(NULL)	高压锅	6800
000B	名古屋	0006	叉子	500
000C	大阪	0006	叉子	500
000B	名古屋	0007	擦菜板	880
000C	大阪	0007	擦菜板	880
(NULL)	(NULL)	(NULL)	圆珠笔	100

我们观察上述结果可以发现, 有两种商品: 高压锅和圆珠笔, 在所有商店都没有销售. 由于我们在 SELECT 子句选择列的显示顺序以及未对结果进行排序的原因, 这个事实需要你仔细地进行观察.

### ● 外连结要点 1: 选取出单张表中全部的信息

与内连结的结果相比, 不同点显而易见, 那就是结果的行数不一样.

内连结的结果中有 13 条记录, 而外连结的结果中有 15 条记录, 增加的 2 条记录到底是什么呢?

这正是外连结的关键点. 多出的 2 条记录是高压锅和圆珠笔, 这 2 条记录在 shopproduct 表中并不存在, 也就是说, 这 2 种商品在任何商店中都没有销售.

由于内连结只能选取出同时存在于两张表中的数据, 因此只在 product 表中存在的 2 种商品并没有出现在结果之中.

相反, 对于外连结来说, 只要数据存在于某一张表当中, 就能够读取出来.

在实际的业务中, 例如想要生成固定行数的单据时, 就需要使用外连结. 如果使用内连结的话, 根据 SELECT 语句执行时商店库存状况的不同, 结果的行数也会发生改变, 生成的单据的版式也会受到影响, 而使用外连结能够得到固定行数的结果.

虽说如此, 那些表中不存在的信息我们还是无法得到, 结果中高压锅和圆珠笔的商店编号和商店名称都是 NULL (具体信息大家都不知道, 真是无可奈何) .

外连结名称的由来也跟 NULL 有关, 即“结果中包含原表中不存在 (在原表之外) 的信息”. 相反, 只包含表内信息的连结也就被称为内连结了.

### ● 外连结要点 2: 使用 LEFT、RIGHT 来指定主表.

外连结还有一点非常重要, 那就是要把哪张表作为主表.

最终的结果中会包含主表内所有的数据. 指定主表的关键字是 **LEFT** 和 **RIGHT**.

顾名思义, 使用 LEFT 时 FROM 子句中写在左侧的表是主表, 使用 RIGHT 时右侧的表是主表.

代码清单 7-11 中使用了 RIGHT, 因此, 右侧的表, 也就是 product 表是主表. 我们还可以像代码清单 7-12 这样进行改写, 意思完全相同. 这样你可能会困惑, 到底应该使用 LEFT 还是 RIGHT?

其实它们的功能没有任何区别, 使用哪一个都可以. 通常使用 LEFT 的情况会多一些, 但也并没有非使用这个不可的理由, 使用 RIGHT 也没有问题.

通过交换两个表的顺序, 同时将 LEFT 更换为 RIGHT(如果原先是 RIGHT, 则更换为 LEFT), 两种方式会到完全相同的结果.

#### 4.2.2.3 结合 WHERE 子句使用左连结

上一小节我们学到了外连结的基础用法, 并且在上一节也学习了结合 WHERE 子句使用内连结的方法, 但在结合 WHERE 子句使用外连结时, 由于外连结的结果很可能与内连结的结果不一样, 会包含那些主表中无法匹配到的行, 并用缺失值填写另一表中的列, 由于这些行的存在, 因此在外连结时使用 WHERE 子句, 情况会有些不一样. 我们来看一个例子:

练习题:

使用外连结从 shopproduct 表和 product 表中找出那些在某个商店库存少于 50 的商品及对应的商店. 希望得到如下结果.

product_id	product_name	sale_price	shop_id	shop_name	quantity
0001	T恤	1000	000A	东京	30
0003	运动T恤	4000	000A	东京	15
0002	打孔器	500	000B	名古屋	30
0004	菜刀	3000	000B	名古屋	20
0006	叉子	500	000B	名古屋	10
0007	擦菜板	880	000B	名古屋	40
0003	运动T恤	4000	000C	大阪	20
0005	高压锅	6800	(Null)	(Null)	(Null)
0008	圆珠笔	100	(Null)	(Null)	(Null)

注意高压锅和圆珠笔两种商品在所有商店都无货, 所以也应该包括在内.

按照 "结合 WHERE 子句使用内连结" 的思路, 我们很自然会写出如下代码

```
1 | SELECT
2 |     P.product_id,
3 |     P.product_name,
```

```

4      P.sale_price,
5      SP.shop_id,
6      SP.shop_name,
7      SP.quantity
8  FROM product AS P
9
10 LEFT OUTER JOIN shopproduct AS SP
11
12 ON SP.product_id = P.product_id
13
14 WHERE quantity < 50

```

然而不幸的是, 得到的却是如下的结果:

product_id	product_name	sale_price	shop_id	shop_name	quantity
0001	T恤	1000	000A	东京	30
0003	运动T恤	4000	000A	东京	15
0002	打孔器	500	000B	名古屋	30
0004	菜刀	3000	000B	名古屋	20
0006	叉子	500	000B	名古屋	10
0007	擦菜板	880	000B	名古屋	40
0003	运动T恤	4000	000C	大阪	20

观察发现, 返回结果缺少了在所有商店都无货的高压锅和圆珠笔。

聪明的你可能很容易想到, 在 WHERE 过滤条件中增加 OR quantity IS NULL 的判断条件, 便可以得到预期结果。然而在实际环境中, 由于数据量大且数据质量并非像我们设想的那样 "干净", 我们并不能容易地意识到缺失值等问题数据的存在, 因此, 还是让我们想一下如何改写我们的查询以使得它能够适应更复杂的真实数据的情形吧。

联系到我们已经掌握了的 SQL 查询的执行顺序(**FROM->WHERE->SELECT**), 我们发现, 问题可能出在筛选条件上, 因为在进行完外连结后才会执行 WHERE 子句, 因此那些主表中无法被匹配到的行就被 WHERE 条件筛选掉了。

明白了这一点, 我们就可以试着把 WHERE 子句挪到外连结之前进行: 先写个子查询, 用来从 shopproduct 表中筛选 quantity<50 的商品, 然后再把这个子查询和主表连结起来。

我们把上述思路写成 SQL 查询语句:

```

1  SELECT
2      P.product_id,
3      P.product_name,
4      P.sale_price,
5      SP.shop_id,
6      SP.shop_name,
7      SP.quantity
8  FROM product AS P

```

```

9
10 LEFT OUTER JOIN-- 先筛选 quantity<50 的商品
11 (SELECT *
12   FROM shopproduct
13  WHERE quantity < 50 ) AS SP
14
15 ON SP.product_id = P.product_id

```

得到的结果如下：

product_id	product_name	sale_price	shop_id	shop_name	quantity
0001	T恤	1000	000A	东京	30
0003	运动T恤	4000	000A	东京	15
0002	打孔器	500	000B	名古屋	30
0004	菜刀	3000	000B	名古屋	20
0006	叉子	500	000B	名古屋	10
0007	擦菜板	880	000B	名古屋	40
0003	运动T恤	4000	000C	大阪	20
0005	高压锅	6800	(Null)	(Null)	(Null)
0008	圆珠笔	100	(Null)	(Null)	(Null)

#### 4.2.2.4 在 MySQL 中实现全外连结

有了对左连结和右连结的了解，就不难理解全外连结的含义了。全外连结本质上就是对左表和右表的所有行都予以保留，能用 ON 关联到的就把左表和右表的内容在一行内显示，不能被关联到的就分别显示，然后把多余的列用缺失值填充。

遗憾的是，MySQL8.0 目前还不支持全外连结，不过我们可以对左连结和右连结的结果进行 UNION 来实现全外连结。

### 4.2.3 多表连结

通常连结只涉及 2 张表，但有时也会出现必须同时连结 3 张以上的表的情况，原则上连结表的数量并没有限制。

#### 4.2.3.1 多表进行内连结

首先创建一个用于三表连结的表 Inventoryproduct。首先我们创建一张用来管理库存商品的表，假设商品都保存在 P001 和 P002 这 2 个仓库之中。

inventory_id	product_id	inventory_quantity
P001	0001	0
P001	0002	120
P001	0003	200
P001	0004	3
P001	0005	0
P001	0006	99
P001	0007	999
P001	0008	200
P002	0001	10
P002	0002	25
P002	0003	34
P002	0004	19
P002	0005	99
P002	0006	0
P002	0007	0
P002	0008	18

建表语句如下:

```

1 CREATE TABLE Inventoryproduct(
2     inventory_id      CHAR(4) NOT NULL,
3     product_id        CHAR(4) NOT NULL,
4     inventory_quantity INTEGER NOT NULL,
5     PRIMARY KEY (inventory_id, product_id));

```

然后插入一些数据:

```

1 --- DML: 插入数据
2 START TRANSACTION;
3 INSERT INTO Inventoryproduct (inventory_id, product_id,
4 inventory_quantity)
5 VALUES ('P001', '0001', 0);
6 INSERT INTO Inventoryproduct (inventory_id, product_id,
7 inventory_quantity)
8 VALUES ('P001', '0002', 120);
9 INSERT INTO Inventoryproduct (inventory_id, product_id,
10 inventory_quantity)
11 VALUES ('P001', '0003', 200);
12 INSERT INTO Inventoryproduct (inventory_id, product_id,
13 inventory_quantity)
14 VALUES ('P001', '0004', 3);
15 INSERT INTO Inventoryproduct (inventory_id, product_id,
16 inventory_quantity)
17 VALUES ('P001', '0005', 0);
18 INSERT INTO Inventoryproduct (inventory_id, product_id,
19 inventory_quantity)
20 VALUES ('P001', '0006', 99);

```

```

21 | INSERT INTO Inventoryproduct (inventory_id, product_id,
22 | inventory_quantity)
23 | VALUES ('P001', '0007', 999);
24 | INSERT INTO Inventoryproduct (inventory_id, product_id,
25 | inventory_quantity)
26 | VALUES ('P001', '0008', 200);
27 | INSERT INTO Inventoryproduct (inventory_id, product_id,
28 | inventory_quantity)
29 | VALUES ('P002', '0001', 10);
30 | INSERT INTO Inventoryproduct (inventory_id, product_id,
31 | inventory_quantity)
32 | VALUES ('P002', '0002', 25);
33 | INSERT INTO Inventoryproduct (inventory_id, product_id,
34 | inventory_quantity)
35 | VALUES ('P002', '0003', 34);
    | INSERT INTO Inventoryproduct (inventory_id, product_id,
    | inventory_quantity)
    | VALUES ('P002', '0004', 19);
    | INSERT INTO Inventoryproduct (inventory_id, product_id,
    | inventory_quantity)
    | VALUES ('P002', '0005', 99);
    | INSERT INTO Inventoryproduct (inventory_id, product_id,
    | inventory_quantity)
    | VALUES ('P002', '0006', 0);
    | INSERT INTO Inventoryproduct (inventory_id, product_id,
    | inventory_quantity)
    | VALUES ('P002', '0007', 0);
    | INSERT INTO Inventoryproduct (inventory_id, product_id,
    | inventory_quantity)
    | VALUES ('P002', '0008', 18);
    | COMMIT;

```

接下来, 我们根据上表及 shopproduct 表和 product 表, 使用内连接找出每个商店都有那些商品, 每种商品的库存总量分别是多少.

```

1 | SELECT
2 |   SP.shop_id,
3 |   SP.shop_name,
4 |   SP.product_id,
5 |   P.product_name,
6 |   P.sale_price,
7 |   IP.inventory_quantity
8 | FROM shopproduct AS SP
9 |
10 | INNER JOIN product AS P
11 |
12 | ON SP.product_id = P.product_id

```

```

13
14 INNER JOIN Inventoryproduct AS IP
15
16 ON SP.product_id = IP.product_id
17
18 WHERE IP.inventory_id = 'P001';

```

得到如下结果

shop_id	shop_name	product_id	product_name	sale_price	inventory_quantity
000A	东京	0001	T恤	1000	0
000A	东京	0002	打孔器	500	120
000A	东京	0003	运动T恤	4000	200
000B	名古屋	0002	打孔器	500	120
000B	名古屋	0003	运动T恤	4000	200
000B	名古屋	0004	菜刀	3000	3
000B	名古屋	0006	叉子	500	99
000B	名古屋	0007	擦菜板	880	999
000C	大阪	0003	运动T恤	4000	200
000C	大阪	0004	菜刀	3000	3
000C	大阪	0006	叉子	500	99
000C	大阪	0007	擦菜板	880	999
000D	福冈	0001	T恤	1000	0

我们可以看到, 连结第三张表的时候, 也是通过 **ON** 子句指定连结条件(这里使用最基础的等号将作为连结条件的 **product** 表和 **shopproduct** 表中的商品编号 **product\_id** 连结了起来), 由于 **product** 表和 **shopproduct** 表已经进行了连结, 因此就无需再对 **product** 表和 **Inventoryproduct** 表进行连结了(虽然也可以进行连结, 但结果并不会发生改变, 因为本质上并没有增加新的限制条件).

即使想要把连结的表增加到 4 张、5 张.....使用 **INNER JOIN** 进行添加的方式也是完全相同的.

#### 4.2.3.2 多表进行外连结

正如之前所学发现的, 外连结一般能比内连结有更多的行, 从而能够比内连结给出更多关于主表的信息, 多表连结的时候使用外连结也有同样的作用.

例如,

```

1 SELECT
2     P.product_id,
3     P.product_name,
4     P.sale_price,
5     SP.shop_id,
6     SP.shop_name,
7     IP.inventory_quantity

```

```

8 | FROM product AS P
9 |
10| LEFT OUTER JOIN shopproduct AS SP
11|
12| ON SP.product_id = P.product_id
13|
14| LEFT OUTER JOIN Inventoryproduct AS IP
15|
16| ON SP.product_id = IP.product_id

```

## 查询结果

product_id	product_name	sale_price	shop_id	shop_name	inventory_quantity
0001	T恤	1000	000A	东京	0
0001	T恤	1000	000D	福冈	0
0002	打孔器	500	000A	东京	120
0002	打孔器	500	000B	名古屋	120
0003	运动T恤	4000	000A	东京	200
0003	运动T恤	4000	000B	名古屋	200
0003	运动T恤	4000	000C	大阪	200
0004	菜刀	3000	000B	名古屋	3
0004	菜刀	3000	000C	大阪	3
0006	叉子	500	000B	名古屋	99
0006	叉子	500	000C	大阪	99
0007	擦菜板	880	000B	名古屋	999
0007	擦菜板	880	000C	大阪	999
0001	T恤	1000	000A	东京	10
0001	T恤	1000	000D	福冈	10
0002	打孔器	500	000A	东京	25
0002	打孔器	500	000B	名古屋	25
0003	运动T恤	4000	000A	东京	34
0003	运动T恤	4000	000B	名古屋	34
0003	运动T恤	4000	000C	大阪	34
0004	菜刀	3000	000B	名古屋	19
0004	菜刀	3000	000C	大阪	19
0006	叉子	500	000B	名古屋	0
0006	叉子	500	000C	大阪	0
0007	擦菜板	880	000B	名古屋	0
0007	擦菜板	880	000C	大阪	0
0005	高压锅	6800	(Null)	(Null)	(Null)
0008	圆珠笔	100	(Null)	(Null)	(Null)

## 4.2.4 ON 子句进阶-非等值连结

在刚开始介绍连结的时候, 书上提到过, 除了使用相等判断的等值连结, 也可以使用比较运算符来进行连接. 实际上, 包括比较运算符 (<, <=, >, >=, BETWEEN) 和谓词运算 (LIKE, IN, NOT 等等) 在内的所有的逻辑运算都可以放在 ON 子句内作为连结条件.

### 4.2.4.1 非等值自左连结(SELF JOIN)

使用非等值自左连结实现排名。

练习题：

希望对 product 表中的商品按照售价赋予排名。一个从集合论出发，使用自左连结的思路是，对每一种商品，找出售价不低于它的所有商品，然后对售价不低于它的商品使用 COUNT 函数计数。例如，对于价格最高的商品，

```
1 | SELECT
2 |     product_id,
3 |     product_name,
4 |     sale_price,
5 |     COUNT(p2_id) AS rank_id
6 | FROM ( -- 使用自左连结对每种商品找出价格不低于它的商品
7 |         SELECT
8 |             P1.product_id,
9 |             P1.product_name,
10 |             P1.sale_price,
11 |             P2.product_id AS P2_id,
12 |             P2.product_name AS P2_name,
13 |             P2.sale_price AS P2_price
14 |         FROM product AS P1
15 |
16 |         LEFT OUTER JOIN product AS P2
17 |
18 |         ON P1.sale_price <= P2.sale_price
19 |     ) AS X
20 |
21 | GROUP BY product_id, product_name, sale_price
22 |
23 | ORDER BY rank_id;
```

注 1：COUNT 函数的参数是列名时，会忽略该列中的缺失值，参数为 \* 时则不忽略缺失值。

注 2：上述排名方案存在一些问题—如果两个商品的价格相等，则会导致两个商品的排名错误，例如，叉子和打孔器的排名应该都是第六，但上述查询导致二者排名都是第七。试修改上述查询使得二者的排名均为第六。

product_id	product_name	sale_price	rank
0005	高压锅	6800	1
0003	运动T恤	4000	2
0004	菜刀	3000	3
0001	T恤	1000	4
0007	擦菜板	880	5
0006	叉子	500	7
0002	打孔器	500	7
0008	圆珠笔	100	8

注 3: 实际上, 进行排名有专门的函数, 这是 MySQL 8.0 新增加的窗口函数中的一种(窗口函数将在下一章学习), 但在较低版本的 MySQL 中只能使用上述自左连结的思路.

使用非等值自左连结进行累计求和:

练习题:

请按照商品的售价从低到高, 对售价进行累计求和[注: 这个案例缺少实际意义, 并且由于有两种商品价格相同导致了不必要的复杂度, 但示例数据库的表结构比较简单, 暂时未想出有实际意义的例题]

首先, 按照题意, 对每种商品使用自左连结, 找出比该商品售价价格更低或相等的商品

```
1  SELECT
2      P1.product_id,
3      P1.product_name,
4      P1.sale_price,
5      P2.product_id AS P2_id,
6      P2.product_name AS P2_name,
7      P2.sale_price AS P2_price
8  FROM product AS P1
9
10 LEFT OUTER JOIN product AS P2
11
12 ON P1.sale_price >= P2.sale_price
13
14 ORDER BY P1.sale_price,P1.product_id
```

查看查询结果

product_id	product_name	sale_price	P2_id	P2_name	P2_price
0008	圆珠笔	100	0008	圆珠笔	100
0002	打孔器	500	0006	叉子	500
0002	打孔器	500	0008	圆珠笔	100
0002	打孔器	500	0002	打孔器	500
0006	叉子	500	0008	圆珠笔	100
0006	叉子	500	0002	打孔器	500
0006	叉子	500	0006	叉子	500
0007	擦菜板	880	0008	圆珠笔	100
0007	擦菜板	880	0006	叉子	500
0007	擦菜板	880	0002	打孔器	500
0007	擦菜板	880	0007	擦菜板	880
0001	T恤	1000	0002	打孔器	500
0001	T恤	1000	0007	擦菜板	880
0001	T恤	1000	0001	T恤	1000
0001	T恤	1000	0006	叉子	500
0001	T恤	1000	0008	圆珠笔	100
0004	菜刀	3000	0004	菜刀	3000
0004	菜刀	3000	0001	T恤	1000
0004	菜刀	3000	0007	擦菜板	880
0004	菜刀	3000	0006	叉子	500
0004	菜刀	3000	0008	圆珠笔	100
0004	菜刀	3000	0002	打孔器	500
0003	运动T恤	4000	0008	圆珠笔	100
0003	运动T恤	4000	0007	擦菜板	880
0003	运动T恤	4000	0001	T恤	1000
0003	运动T恤	4000	0006	叉子	500
0003	运动T恤	4000	0004	菜刀	3000
0003	运动T恤	4000	0003	运动T恤	4000
0003	运动T恤	4000	0002	打孔器	500
0005	高压锅	6800	0006	叉子	500
0005	高压锅	6800	0007	擦菜板	880
0005	高压锅	6800	0005	高压锅	6800
0005	高压锅	6800	0004	菜刀	3000
0005	高压锅	6800	0003	运动T恤	4000
0005	高压锅	6800	0008	圆珠笔	100
0005	高压锅	6800	0002	打孔器	500
0005	高压锅	6800	0001	T恤	1000

看起来似乎没什么问题.

下一步, 按照 P1.product\_Id 分组, 对 P2\_price 求和:

```

1  SELECT
2      product_id,
3      product_name,
4      sale_price,
5      SUM(P2_price) AS cum_price
6
7  FROM (SELECT
8      P1.product_id,
```

```

9      P1.product_name,
10     P1.sale_price,
11     P2.product_id AS P2_id,
12     P2.product_name AS P2_name,
13     P2.sale_price AS P2_price
14   FROM product AS P1
15   LEFT OUTER JOIN product AS P2
16   ON P1.sale_price >= P2.sale_price
17   ORDER BY P1.sale_price,P1.product_id ) AS X
18
19 GROUP BY product_id, product_name, sale_price
20
21 ORDER BY sale_price,product_id;

```

得到的查询结果为：

product_id	product_name	sale_price	cum_price
0008	圆珠笔	100	100
0002	打孔器	500	1100
0006	叉子	500	1100
0007	擦菜板	880	1980
0001	T恤	1000	2980
0004	菜刀	3000	5980
0003	运动T恤	4000	9980
0005	高压锅	6800	16780

观察上述查询结果发现,由于有两种商品的售价相同,在使用  $\geq$  进行连结时,导致了累计求和错误,这是由于这两种商品售价相同导致的.因此实际上之前是不应该单独只用  $\geq$  作为连结条件的.

考察我们建立自左连结的本意,是要找出满足:

1. 比该商品售价更低的,或者是
2. 该种商品自身,以及
3. 如果 A 和 B 两种商品售价相等,则建立连结时,如果 P1.A 和 P2.A,P2.B 建立了连接,则 P1.B 不再和 P2.A 建立连结,因此根据上述约束条件,利用 ID 的有序性,进一步将上述查询改写为:

```

1  SELECT
2      product_id,
3      product_name,
4      sale_price,
5      SUM(P2_price) AS cum_price
6  FROM
7  (SELECT
8          P1.product_id,
9          P1.product_name,

```

```

10    P1.sale_price,
11    P2.product_id AS P2_id,
12    P2.product_name AS P2_name,
13    P2.sale_price AS P2_price
14  FROM product AS P1
15
16  LEFT OUTER JOIN product AS P2
17  ON ((P1.sale_price > P2.sale_price)
18      OR (P1.sale_price = P2.sale_price
19          AND P1.product_id<=P2.product_id))
20  ORDER BY P1.sale_price,P1.product_id) AS X
21
22 GROUP BY product_id, product_name, sale_price
23
24 ORDER BY sale_price,cum_price;

```

这次结果就正确了.

product_id	product_name	sale_price	cum_price
0008	圆珠笔	100	100
0006	叉子	500	600
0002	打孔器	500	1100
0007	擦菜板	880	1980
0001	T恤	1000	2980
0004	菜刀	3000	5980
0003	运动T恤	4000	9980
0005	高压锅	6800	16780

## 4.2.5 交叉连结——CROSS JOIN(笛卡尔积)

之前的无论是外连结内连结, 一个共同的必备条件就是连结条件 -ON 子句, 用来指定连结的条件. 如果你试过不使用这个连结条件的连结查询, 你可能已经发现, 结果会有很多行. 在连结去掉 ON 子句, 就是所谓的交叉连结 (CROSS JOIN), 交叉连结又叫笛卡尔积, 后者是一个数学术语. 两个集合做笛卡尔积, 就是使用集合 A 中的每一个元素与集合 B 中的每一个元素组成一个有序的组合.

数据库表(或者子查询) 的并, 交和差都是在纵向上对表进行扩张或筛选限制等运算的, 这要求表的列数及对应位置的列的数据类型 "相容", 因此这些运算并不会增加新的列, 而交叉连接 (笛卡尔积) 则是在横向上对表进行扩张, 即增加新的列, 这一点和连结的功能是一致的. 但因为没有了 ON 子句的限制, 会对左表和右表的每一行进行组合, 这经常会导致很多无意义的行出现在检索结果中. 当然, 在某些查询需求中, 交叉连结也有一些用处.

交叉连结的语法有如下几种形式:

```

1 -- 1. 使用关键字 CROSS JOIN 显式地进行交叉连结
2

```

```

3 | SELECT
4 |     SP.shop_id,
5 |     SP.shop_name,
6 |     SP.product_id,
7 |     P.product_name,
8 |     P.sale_price
9 |
10| FROM shopproduct AS SP
11|
12| CROSS JOIN product AS P;
13|
14--2. 使用逗号分隔两个表，并省略 ON 子句
15|
16| SELECT
17|     SP.shop_id,
18|     SP.shop_name,
19|     SP.product_id,
20|     P.product_name,
21|     P.sale_price
22| FROM shopproduct AS SP ,product AS P;

```

请大家试着执行一下以上语句.

可能大家会惊讶于结果的行数, 但我们还是先来介绍一下语法结构吧. 对满足相同规则的表进行交叉连结的集合运算符是**CROSS JOIN** (笛卡儿积) . 进行交叉连结时无法使用内连结和外连结中所使用的 ON 子句, 这是因为交叉连结是对两张表中的全部记录进行交叉组合, 因此结果中的记录数通常是两张表中行数的乘积. 本例中, 因为 shopproduct 表存在 13 条记录, product 表存在 8 条记录, 所以结果中就包含了  **$13 \times 8 = 104$**  条记录.

可能这时会有读者想起前面我们提到过集合运算中的乘法会在本节中进行详细学习, 这就是上面介绍的交叉连结. 内连结是交叉连结的一部分, “内”也可以理解为“包含在交叉连结结果中的部分”. 相反, 外连结的“外”可以理解为“交叉连结结果之外的部分”.

交叉连结没有应用到实际业务之中的原因有两个. 一是其结果没有实用价值, 二是由于其结果行数太多, 需要花费大量的运算时间和高性能设备的支持.

#### 4.2.5.1 [扩展阅读]连结与笛卡儿积的关系

考察笛卡儿积和连结, 不难发现, 笛卡儿积可以视作一种特殊的连结(事实上笛卡儿积的语法也可以写作 CROSS JOIN), 这种连结的 ON 子句是一个恒为真的谓词.

反过来思考, 在对笛卡儿积进行适当的限制之后, 也就得到了内连结和外连结.

例如, 对于 shopproduct 表和 product 表, 首先建立笛卡尔乘积:

```
1 | SELECT SP.* , P.*
```

```

2 | FROM shopproduct AS SP
3 |
4 | CROSS JOIN product AS P;
5 |

```

查询结果的一部分如下：

shop_id	shop_name	product_id	quantity	product_id(1)	product_name	product_type	sale_price	purchase_price	regist_date
000A	东京	0001	30	0008	圆珠笔	办公用品	100	Null	2009-11-11
000A	东京	0001	30	0007	擦菜板	厨房用具	880	790	2008-04-28
000A	东京	0001	30	0006	叉子	厨房用具	500	Null	2009-09-20
000A	东京	0001	30	0005	高压锅	厨房用具	6800	5000	2009-01-15
000A	东京	0001	30	0004	菜刀	厨房用具	3000	2800	2009-09-20
000A	东京	0001	30	0003	运动T恤	衣服	4000	2800	Null
000A	东京	0001	30	0002	打孔器	办公用品	500	320	2009-09-11
000A	东京	0001	30	0001	T恤	衣服	1000	500	2009-09-20
000A	东京	0002	50	0008	圆珠笔	办公用品	100	Null	2009-11-11
000A	东京	0002	50	0007	擦菜板	厨房用具	880	790	2008-04-28
000A	东京	0002	50	0006	叉子	厨房用具	500	Null	2009-09-20
000A	东京	0002	50	0005	高压锅	厨房用具	6800	5000	2009-01-15
000A	东京	0002	50	0004	菜刀	厨房用具	3000	2800	2009-09-20
000A	东京	0002	50	0003	运动T恤	衣服	4000	2800	Null
000A	东京	0002	50	0002	打孔器	办公用品	500	320	2009-09-11
000A	东京	0002	50	0001	T恤	衣服	1000	500	2009-09-20
000A	东京	0003	15	0008	圆珠笔	办公用品	100	Null	2009-11-11
000A	东京	0003	15	0007	擦菜板	厨房用具	880	790	2008-04-28
000A	东京	0003	15	0006	叉子	厨房用具	500	Null	2009-09-20
000A	东京	0003	15	0005	高压锅	厨房用具	6800	5000	2009-01-15
000A	东京	0003	15	0004	菜刀	厨房用具	3000	2800	2009-09-20
000A	东京	0003	15	0003	运动T恤	衣服	4000	2800	Null
000A	东京	0003	15	0002	打孔器	办公用品	500	320	2009-09-11
000A	东京	0003	15	0001	T恤	衣服	1000	500	2009-09-20
000B	名古屋	0002	30	0008	圆珠笔	办公用品	100	Null	2009-11-11
000B	名古屋	0002	30	0007	擦菜板	厨房用具	880	790	2008-04-28
000B	名古屋	0002	30	0006	叉子	厨房用具	500	Null	2009-09-20
000B	名古屋	0002	30	0005	高压锅	厨房用具	6800	5000	2009-01-15
000B	名古屋	0002	30	0004	菜刀	厨房用具	3000	2800	2009-09-20
000B	名古屋	0002	30	0003	运动T恤	衣服	4000	2800	Null
000B	名古屋	0002	30	0002	打孔器	办公用品	500	320	2009-09-11
000B	名古屋	0002	30	0001	T恤	衣服	1000	500	2009-09-20
000B	名古屋	0003	120	0008	圆珠笔	办公用品	100	Null	2009-11-11
000B	名古屋	0003	120	0007	擦菜板	厨房用具	880	790	2008-04-28
000B	名古屋	0003	120	0006	叉子	厨房用具	500	Null	2009-09-20
000B	名古屋	0003	120	0005	高压锅	厨房用具	6800	5000	2009-01-15
000B	名古屋	0003	120	0004	菜刀	厨房用具	3000	2800	2009-09-20
000B	名古屋	0003	120	0003	运动T恤	衣服	4000	2800	Null
000B	名古屋	0003	120	0002	打孔器	办公用品	500	320	2009-09-11
000B	名古屋	0003	120	0001	T恤	衣服	1000	500	2009-09-20
000B	名古屋	0004	20	0008	圆珠笔	办公用品	100	Null	2009-11-11
000B	名古屋	0004	20	0007	擦菜板	厨房用具	880	790	2008-04-28
000B	名古屋	0004	20	0006	叉子	厨房用具	500	Null	2009-09-20
000B	名古屋	0004	20	0005	高压锅	厨房用具	6800	5000	2009-01-15
000B	名古屋	0004	20	0004	菜刀	厨房用具	3000	2800	2009-09-20
000B	名古屋	0004	20	0003	运动T恤	衣服	4000	2800	Null
000B	名古屋	0004	20	0002	打孔器	办公用品	500	320	2009-09-11
000B	名古屋	0004	20	0001	T恤	衣服	1000	500	2009-09-20
000B	名古屋	0006	10	0008	圆珠笔	办公用品	100	Null	2009-11-11
000B	名古屋	0006	10	0007	擦菜板	厨房用具	880	790	2008-04-28
000B	名古屋	0006	10	0006	叉子	厨房用具	500	Null	2009-09-20
000B	名古屋	0006	10	0005	高压锅	厨房用具	6800	5000	2009-01-15
000B	名古屋	0006	10	0004	菜刀	厨房用具	3000	2800	2009-09-20
000B	名古屋	0006	10	0003	运动T恤	衣服	4000	2800	Null
000B	名古屋	0006	10	0002	打孔器	办公用品	500	320	2009-09-11
000B	名古屋	0006	10	0001	T恤	衣服	1000	500	2009-09-20
000B	名古屋	0007	40	0008	圆珠笔	办公用品	100	Null	2009-11-11

然后对上述笛卡尔乘积增加筛选条件 `SP.product_id=P.product_id`, 就得到了和内连结一致的结果:

```

| SELECT SP.* , P.*

```

```

1   FROM shopproduct AS SP
2
3   CROSS JOIN product AS P
4
5
6 WHERE SP.product_id = P.product_id;
7

```

查询结果如下:

shop_id	shop_name	product_id	quantity	product_id(1)	product_name	product_type	sale_price	purchase_price	regist_date
000A	东京	0001	30	0001	T恤	衣服	1000	500	2009-09-20
000A	东京	0002	50	0002	打孔器	办公用品	500	320	2009-09-11
000A	东京	0003	15	0003	运动T恤	衣服	4000	2800	(Null)
000B	名古屋	0002	30	0002	打孔器	办公用品	500	320	2009-09-11
000B	名古屋	0003	120	0003	运动T恤	衣服	4000	2800	(Null)
000B	名古屋	0004	20	0004	菜刀	厨房用具	3000	2800	2009-09-20
000B	名古屋	0006	10	0006	叉子	厨房用具	500	(Null)	2009-09-20
000B	名古屋	0007	40	0007	擦菜板	厨房用具	880	790	2008-04-28
000C	大阪	0003	20	0003	运动T恤	衣服	4000	2800	(Null)
000C	大阪	0004	50	0004	菜刀	厨房用具	3000	2800	2009-09-20
000C	大阪	0006	90	0006	叉子	厨房用具	500	(Null)	2009-09-20
000C	大阪	0007	70	0007	擦菜板	厨房用具	880	790	2008-04-28
000D	福冈	0001	100	0001	T恤	衣服	1000	500	2009-09-20

实际上,正如书中所说,上述写法中,将 CROSS JOIN 改为逗号后,正是内连结的旧式写法,但在 ANSI 和 ISO 的 SQL-92 标准中,已经将使用 INNER JION ...ON... 的写法规定为标准写法,因此极力推荐大家在平时写 SQL 查询时,使用规范写法.

#### 4.2.6 连结的特定语法和过时语法

在笛卡尔积的基础上,我们增加一个 WHERE 子句,将之前的连结条件作为筛选条件加进去,我们会发现,得到的结果恰好是直接使用内连接的结果.

试执行以下查询,并将查询结果与内连结一节第一个例子的结果做对比.

```

1   SELECT
2       SP.shop_id,
3       SP.shop_name,
4       SP.product_id,
5       P.product_name,
6       P.sale_price
7   FROM shopproduct AS SP
8   CROSS JOIN product AS P
9   WHERE SP.product_id = P.product_id;

```

我们发现,这两个语句得到的结果是相同的.

之前我们学习的内连结和外连结的语法都符合标准 SQL 的规定,可以在所有 DBMS 中执行,因此大家可以放心使用.但是如果大家之后从事系统开发工作,或者阅读遗留 SQL 查询语句的话,一定会碰到需要阅读他人写的代码并进行维护的情况,而那些使用特定和过时语法的程序就会成为我们的麻烦.

SQL 是一门特定语法及过时语法非常多的语言, 虽然之前本书中也多次提及, 但连结是其中特定语法的部分, 现在还有不少年轻的程序员和系统工程师仍在使用这些特定的语法. 例如, 将本节最初介绍的内连结的 SELECT 语句替换为过时语法的结果如下所示.

使用过时语法的内连结 (结果与代码清单 7-9 相同)

```
1 | SELECT
2 |     SP.shop_id,
3 |     SP.shop_name,
4 |     SP.product_id,
5 |     P.product_name,
6 |     P.sale_price
7 | FROM shopproduct SP,product P
8 | WHERE SP.product_id = P.product_id AND SP.shop_id = '000A';
```

这样的书写方式所得到的结果与标准语法完全相同, 并且这样的语法可以在所有的 DBMS 中执行, 并不能算是特定的语法, 只是过时了而已.

但是, 由于这样的语法不仅过时, 而且还存在很多其他的问题, 因此不推荐大家使用, 理由主要有以下三点:

第一, 使用这样的语法无法马上判断出到底是内连结还是外连结 (又或者是其他种类的连结) .

第二, 由于连结条件都写在 WHERE 子句之中, 因此无法在短时间内分辨出哪部分是连结条件, 哪部分是用来选取记录的限制条件.

第三, 我们不知道这样的语法到底还能使用多久. 每个 DBMS 的开发者都会考虑放弃过时的语法, 转而支持新的语法. 虽然并不是马上就不能使用了, 但那一天总会到来的.

虽然这么说, 但是现在使用这些过时语法编写的程序还有很多, 到目前为止还都能正常执行. 我想大家很可能会碰到这样的代码, 因此还是希望大家能够了解这些知识.

## 练习题

### 练习题 4.1

找出 product 和 product2 中售价高于 500 的商品的基本信息。

### 练习题 4.2

借助对称差的实现方式, 求 product 和 product2 的交集。

### **练习题 4.3**

每类商品中售价最高的商品都在哪些商店有售？

### **练习题 4.4**

分别使用内连结和关联子查询每一类商品中售价最高的商品。

### **练习题 4.5**

用关联子查询实现：在 `product` 表中，取出 `product_id`, `product_name`, `sale_price`, 并按照商品的售价从低到高进行排序、对售价进行累计求和。