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/aicampsql

Task03：复杂查询方法 - 视图、子查询、函数等

3.1 视图

我们先来看一个查询语句（仅做示例，未提供相关数据）

```
SELECT stu_name FROM view_students_info;
```

单从表面上看起来这个语句是和正常的从数据表中查询数据是完全相同的，但其实我们操作的是一个视图。所以从 SQL 的角度来说操作视图与操作表看起来是完全相同的，那么为什么还会有视图的存在呢？视图到底是什么？视图与表有什么不同呢？

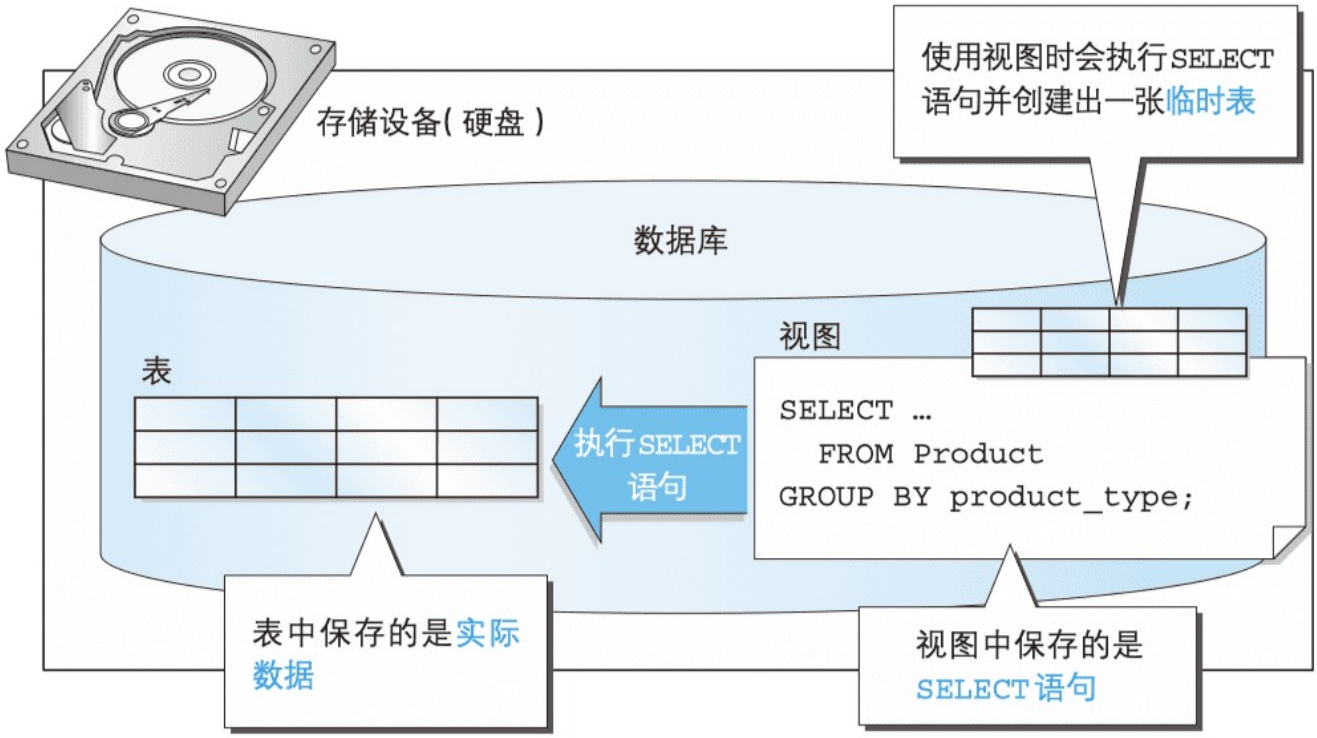
3.1.1 什么是视图

视图是一个虚拟的表，不同于直接操作数据表，视图是依据 SELECT 语句来创建的（会在下面具体介绍），所以操作视图时会根据创建视图的 SELECT 语句生成一张虚拟表，然后在这张虚拟表上做 SQL 操作。

3.1.2 视图与表有什么区别

《sql 基础教程第 2 版 》用一句话非常凝练的概括了视图与表的区别—“是否保存了实际的数据”。所以视图并不是数据库真实存储的数据表，它可以看作是一个窗口，通过这个窗口我们可以看到数据库表中真实存在的数据。所以我们要区别视图和数据表的本质，即视图是

基于真实表的一张虚拟的表，其数据来源均建立在真实表的基础上。



图片来源：《sql 基础教程第 2 版》

下面这句顺口溜也方便大家记忆视图与表的关系：**“视图不是表，视图是虚表，视图依赖于表，视图不保存正式数据”**。

3.1.3 为什么会存在视图

那既然已经有数据表了，为什么还需要视图呢？主要有以下几点原因：

1. 通过定义视图可以将频繁使用的 SELECT 语句保存以提高效率。
2. 通过定义视图可以使用户看到的数据更加清晰。
3. 通过定义视图可以不对外公开数据表全部字段，增强数据的保密性。
4. 通过定义视图可以降低数据的冗余。

3.1.4 如何创建视图

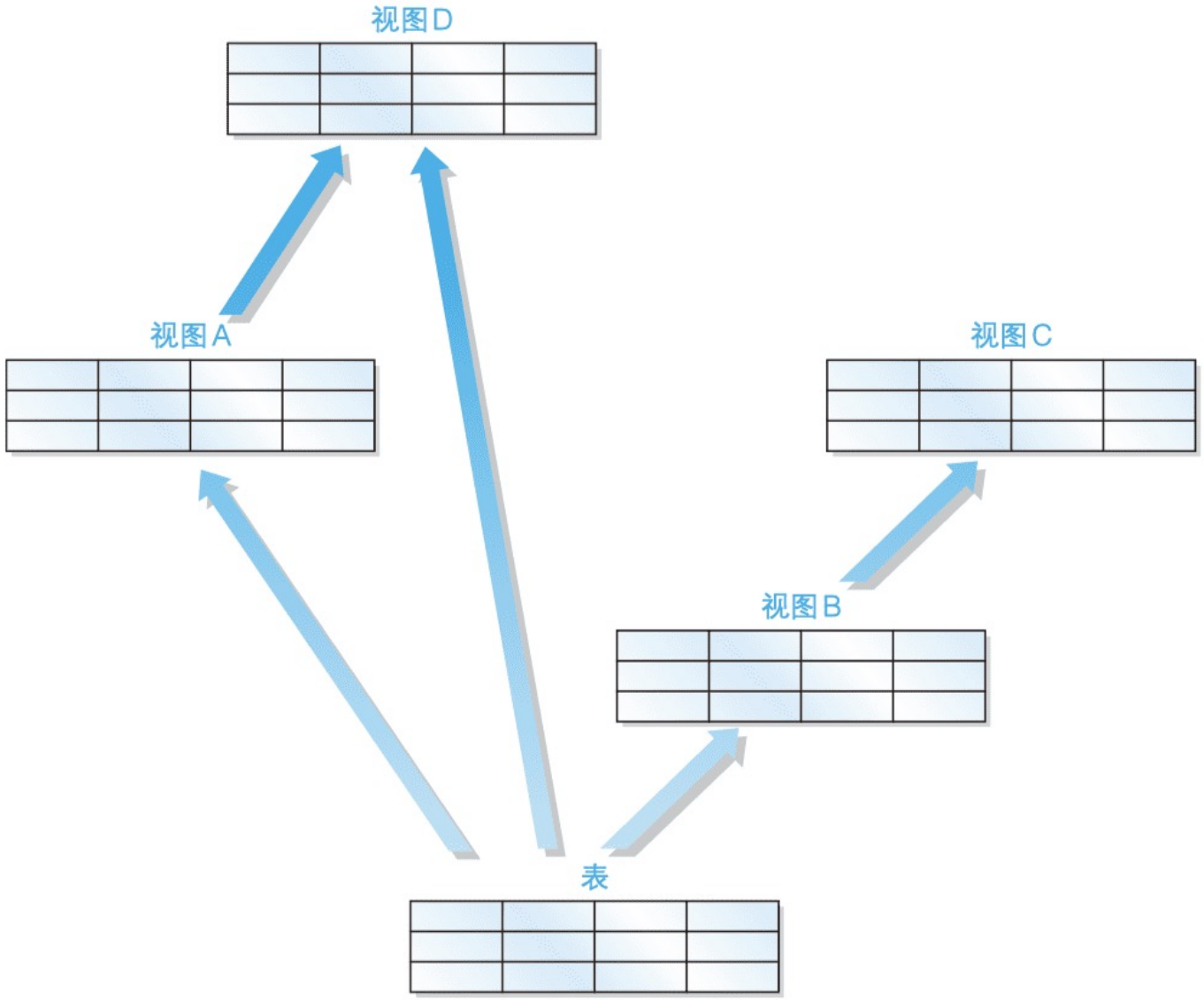
说了这么多视图与表的区别，下面我们就一起来看一下如何创建视图吧。创建视图的基本语法如下：

```
CREATE VIEW < 视图名称 >(< 列名 1>,< 列名 2>,...) AS <SELECT 语句 >
```

其中 SELECT 语句需要书写在 AS 关键字之后。 SELECT 语句中列的排列顺序和视图中列的排列顺序相同， SELECT 语句中的第 1 列就是视图中的第 1 列， SELECT 语句中的第 2 列

就是视图中的第 2 列，以此类推。而且视图的列名是在视图名称之后的列表中定义的。需要注意的是视图名在数据库中需要是唯一的，不能与其他视图和表重名。

视图不仅可以基于真实表，我们也可以在视图的基础上继续创建视图。



图片来源：《sql 基础教程第 2 版》

虽然在视图上继续创建视图的语法没有错误，但是我们还是应该尽量避免这种操作。这是因为对多数 DBMS 来说， 多重视图会降低 SQL 的性能。

- 注意事项

需要注意的是在一般的 DBMS 中定义视图时不能使用 ORDER BY 语句。下面这样定义视图是错误的。

```
CREATE VIEW productsum (product_type, cnt_product)
AS
SELECT product_type, COUNT(*)
FROM product
```

```
GROUP BY product_type
ORDER BY product_type;
```

为什么不能使用 ORDER BY 子句呢？这是因为视图和表一样，**数据行都是没有顺序的**。

在 MySQL 中视图的定义是允许使用 ORDER BY 语句的，但是若从特定视图进行选择，而该视图使用了自己的 ORDER BY 语句，则视图定义中的 ORDER BY 将被忽略。

3.1.4.1 基于单表的视图

我们在 product 表的基础上创建一个视图，如下：

```
CREATE VIEW productsum (product_type, cnt_product)
AS
SELECT product_type, COUNT(*)
FROM product
GROUP BY product_type ;
```

创建的视图如下图所示：

	product_type	cnt_product
1	衣服	2
2	办公用品	2
3	厨房用具	4

3.1.4.2 基于多表的视图

为了学习多表视图，我们再创建一张表，相关代码如下：

```
CREATE TABLE shop_product(
    shop_id    CHAR(4)       NOT NULL,
    shop_name  VARCHAR(200)  NOT NULL,
    product_id CHAR(4)       NOT NULL,
    quantity   INTEGER       NOT NULL,
    PRIMARY KEY (shop_id, product_id)
);

INSERT INTO shop_product (shop_id, shop_name, product_id, quantity)
VALUES ('000A', '东京',      '0001', 30);
INSERT INTO shop_product (shop_id, shop_name, product_id, quantity)
VALUES ('000A', '东京',      '0002', 50);
INSERT INTO shop_product (shop_id, shop_name, product_id, quantity)
VALUES ('000A', '东京',      '0003', 15);
INSERT INTO shop_product (shop_id, shop_name, product_id, quantity)
```

```
VALUES ('000B', '名古屋',    '0002', 30);
INSERT INTO shop_product (shop_id, shop_name, product_id, quantity)
VALUES ('000B', '名古屋',    '0003', 120);
INSERT INTO shop_product (shop_id, shop_name, product_id, quantity)
VALUES ('000B', '名古屋',    '0004', 20);
INSERT INTO shop_product (shop_id, shop_name, product_id, quantity)
VALUES ('000B', '名古屋',    '0006', 10);
INSERT INTO shop_product (shop_id, shop_name, product_id, quantity)
VALUES ('000B', '名古屋',    '0007', 40);
INSERT INTO shop_product (shop_id, shop_name, product_id, quantity)
VALUES ('000C', '大阪',      '0003', 20);
INSERT INTO shop_product (shop_id, shop_name, product_id, quantity)
VALUES ('000C', '大阪',      '0004', 50);
INSERT INTO shop_product (shop_id, shop_name, product_id, quantity)
VALUES ('000C', '大阪',      '0006', 90);
INSERT INTO shop_product (shop_id, shop_name, product_id, quantity)
VALUES ('000C', '大阪',      '0007', 70);
INSERT INTO shop_product (shop_id, shop_name, product_id, quantity)
VALUES ('000D', '福冈',      '0001', 100);
```

我们在 product 表和 shop_product 表的基础上创建视图。

```
CREATE VIEW view_shop_product(product_type, sale_price, shop_name)
AS
SELECT product_type, sale_price, shop_name
FROM product,shop_product
WHERE product.product_id = shop_product.product_id;
```

创建的视图如下图所示

	product_type	sale_price	shop_name
1	衣服	1000	东京
2	办公用品	500	东京
3	衣服	4000	东京
4	办公用品	500	名古屋
5	衣服	4000	名古屋
6	厨房用具	3000	名古屋
7	厨房用具	500	名古屋
8	厨房用具	880	名古屋
9	衣服	4000	大阪
10	厨房用具	3000	大阪
11	厨房用具	500	大阪
12	厨房用具	880	大阪
13	衣服	1000	福冈

我们可以在这个视图的基础上进行查询

```
SELECT sale_price, shop_name
FROM view_shop_product
WHERE product_type = '衣服';
```

查询结果为：

	sale_price	shop_name
1	1000	东京
2	4000	东京
3	4000	名古屋
4	4000	大阪
5	1000	福冈

3.1.5 如何修改视图结构

修改视图结构的基本语法如下：

```
ALTER VIEW < 视图名 > AS <SELECT 语句 >
```

其中视图名在数据库中需要是唯一的，不能与其他视图和表重名。

当然也可以通过将当前视图删除然后重新创建的方式达到修改的效果。（对于数据库底层是不是也是这样操作的呢，你可以自己探索一下。）

- 修改视图

我们修改上方的 productSum 视图为

```
ALTER VIEW productSum
AS
SELECT product_type, sale_price
FROM Product
WHERE regist_date > '2009-09-11';
```

此时 productSum 视图内容如下图所示

	product_type	sale_price
1	衣服	1000
2	厨房用具	3000
3	厨房用具	500
4	办公用品	100

3.1.6 如何更新视图内容

因为视图是一个虚拟表，所以对视图的操作就是对底层基础表的操作，所以在修改时只有满足底层基本表的定义才能成功修改。

对于一个视图来说，如果包含以下结构的任意一种都是不可以被更新的：

- 聚合函数 SUM()、MIN()、MAX()、COUNT() 等
- DISTINCT 关键字
- GROUP BY 子句
- HAVING 子句
- UNION 或 UNION ALL 运算符
- FROM 子句中包含多个表

视图归根结底还是从表派生出来的，因此，如果原表可以更新，那么视图中的数据也可以更新。反之亦然，如果视图发生了改变，而原表没有进行相应更新的话，就无法保证数据的一致性了。

- 更新视图

因为我们刚刚修改的 productSum 视图不包括以上的限制条件，我们来尝试更新一下视图

```sql
UPDATE productsum
SET sale_price = '5000'
WHERE product_type = '办公用品';
```

此时我们再查看 productSum 视图，可以发现数据已经更新了

	product_type	sale_price
1	衣服	1000
2	厨房用具	3000
3	厨房用具	500
4	办公用品	5000

此时观察原表也可以发现数据也被更新了

	product_id	product_name	product_type	sale_price	purchase_price	regist_date
1	0001	T恤	衣服	1000	500	2009-09-20
2	0002	打孔器	办公用品	500	320	2009-09-11
3	0003	运动T恤	衣服	4000	2800	<null>
4	0004	菜刀	厨房用具	3000	2800	2009-09-20
5	0005	高压锅	厨房用具	6800	5000	2009-01-15
6	0006	叉子	厨房用具	500	<null>	2009-09-20
7	0007	擦菜板	厨房用具	880	790	2008-04-28
8	0008	圆珠笔	办公用品	5000	<null>	2009-11-11

不知道大家看到这个结果会不会有疑问，刚才修改视图的时候是设置 product_type='办公用品' 的商品的 sale_price=5000，为什么原表的数据只有一条做了修改呢?

还是因为视图的定义，视图只是原表的一个窗口，所以它修改也只能修改透过窗口能看到的内容。

注意：这里虽然修改成功了，但是并不推荐这种使用方式。而且我们在创建视图时也尽量使用限制不允许通过视图来修改表

3.1.7 如何删除视图

删除视图的基本语法如下：

```sql
DROP VIEW <视图名 1> [, <视图名 2> …]
```

注意：需要有相应的权限才能成功删除。

- 删除视图

我们删除刚才创建的 productSum 视图

```sql
DROP VIEW productSum;
```

如果我们继续操作这个视图的话就会提示当前操作的内容不存在。

3.2 子查询

我们先来看一个语句（仅做示例，未提供相关数据）

```sql
SELECT stu_name
FROM (SELECT stu_name, COUNT(*) AS stu_cnt
        FROM students_info
        GROUP BY stu_age) AS studentSum;
```

这个语句看起来很好理解，其中使用括号括起来的 SQL 语句首先执行，执行成功后再执行外面的 SQL 语句。但是我们上一节提到的视图也是根据 SELECT 语句创建视图然后在这个基础上再进行查询。那么什么是子查询呢？子查询和视图又有什么关系呢？

3.2.1 什么是子查询

子查询指一个查询语句嵌套在另一个查询语句内部的查询，这个特性从 MySQL 4.1 开始引入，在 SELECT 子句中**先计算子查询**，子查询结果作为外层另一个查询的过滤条件，查询可以基于一个表或者多个表。

3.2.2 子查询和视图的关系

子查询就是将用来定义视图的 SELECT 语句直接用于 FROM 子句当中。其中 AS studentSum 可以看作是子查询的名称，而且由于子查询是一次性的，所以**子查询不会像视图那样保存在存储介质中**， 而是在 SELECT 语句执行之后就消失了。

3.2.3 嵌套子查询

与在视图上再定义视图类似，子查询也没有具体的限制，例如我们可以这样

```sql
SELECT product_type, cnt_product
FROM (SELECT *
                FROM (
                        SELECT product_type,
                        COUNT(*) AS cnt_product
                        FROM product
                        GROUP BY product_type
                     ) AS productsum
                WHERE cnt_product = 4
```

```
        ) AS productsum2;
```

其中最内层的子查询我们将其命名为 productSum，这条语句根据 product_type 分组并查询个数，第二层查询中将个数为 4 的商品查询出来，最外层查询 product_type 和 cnt_product 两列。

虽然嵌套子查询可以查询出结果，但是随着子查询嵌套的层数的叠加，SQL 语句不仅会难以理解而且执行效率也会很差，所以要尽量避免这样的使用。

3.2.4 标量子查询

标量就是单一的意思，那么标量子查询也就是单一的子查询，那什么叫做单一的子查询呢?

所谓单一就是要求我们执行的 SQL 语句只能返回一个值，也就是要返回表中具体的 **某一行的某一列**。例如我们有下面这样一张表

```
product_id | product_name | sale_price
-----------+--------------+-----------
0003       | 运动 T 恤      | 4000
0004       | 菜刀           | 3000
0005       | 高压锅         | 6800
```

那么我们执行一次标量子查询后是要返回类似于，“0004”，“菜刀”这样的结果。

3.2.5 标量子查询有什么用

我们现在已经知道标量子查询可以返回一个值了，那么它有什么作用呢?

直接这样想可能会有些困难，让我们看几个具体的需求：

1. 查询出销售单价高于平均销售单价的商品
2. 查询出注册日期最晚的那个商品

你有思路了吗?

让我们看如何通过标量子查询语句查询出销售单价高于平均销售单价的商品。

```
SELECT product_id, product_name, sale_price
FROM product
WHERE sale_price > (
    SELECT AVG(sale_price) FROM product
);
```

上面的这条语句首先后半部分查询出 product 表中的平均售价，前面的 sql 语句在根据

WHERE 条件挑选出合适的商品。
由于标量子查询的特性，导致标量子查询不仅仅局限于 WHERE 子句中，通常任何可以使用单一值的位置都可以使用。也就是说， 能够使用常数或者列名的地方，无论是 SELECT 子句、GROUP BY 子句、HAVING 子句，还是 ORDER BY 子句，几乎所有的地方都可以使用。

我们还可以这样使用标量子查询：

```
SELECT product_id,
       product_name,
       sale_price,
       (SELECT AVG(sale_price) FROM product) AS avg_price
FROM product;
```

你能猜到这段代码的运行结果是什么吗？运行一下看看与你想象的结果是否一致。

3.2.6 关联子查询

- 什么是关联子查询

关联子查询既然包含关联两个字那么一定意味着查询与子查询之间存在着联系。这种联系是如何建立起来的呢？

我们先看一个例子：

```
SELECT product_type, product_name, sale_price
FROM product AS p1
WHERE sale_price > (SELECT AVG(sale_price)
                    FROM product AS p2
                    WHERE p1.product_type = p2.product_type
                    GROUP BY product_type);
```

你能理解这个例子在做什么操作么？先来看一下这个例子的执行结果

	product_type	product_name	sale_price
1	办公用品	打孔器	500
2	衣服	运动T恤	4000
3	厨房用具	菜刀	3000
4	厨房用具	高压锅	6800

通过上面的例子我们大概可以猜到吗，关联子查询就是通过一些标志将内外两层的查询连接起来起到过滤数据的目的，接下来我们就一起看一下关联子查询的具体内容吧。

- 关联子查询与子查询的联系

还记得我们之前的那个例子么 `查询出销售单价高于平均销售单价的商品`，这个例子的 SQL 语句如下

```
SELECT product_id, product_name, sale_price
FROM product
WHERE sale_price > (SELECT AVG(sale_price) FROM product);
```

我们再来看一下这个需求 `选取出各商品种类中高于该商品种类的平均销售单价的商品` 。SQL 语句如下：

```
SELECT product_type, product_name, sale_price
FROM product ASp1
WHERE sale_price > (SELECT AVG(sale_price)
                    FROM product ASp2
                    WHERE p1.product_type =p2.product_type
                    GROUP BY product_type);
```

可以看出上面这两个语句的区别吗?
在第二条 SQL 语句也就是关联子查询中我们将外面的 product 表标记为 p1，将内部的 product 设置为 p2，而且通过 WHERE 语句连接了两个查询。

但是如果刚接触的话一定会比较疑惑关联查询的执行过程，这里有一个 博客 讲的比较清楚。在这里我们简要的概括为：

1. 首先执行不带 WHERE 的主查询
2. 根据主查询讯结果匹配 product_type，获取子查询结果
3. 将子查询结果再与主查询结合执行完整的 SQL 语句

_ 在子查询中像标量子查询，嵌套子查询或者关联子查询可以看作是子查询的一种操作方式即可。_

小结

视图和子查询是数据库操作中较为基础的内容，对于一些复杂的查询需要使用子查询加一些条件语句组合才能得到正确的结果。但是无论如何对于一个 SQL 语句来说都不应该设计的层数非常深且特别复杂，不仅可读性差而且执行效率也难以保证，所以尽量有简洁的语句来完成需要的功能。

练习题 - 第一部分

练习 3.1

创建出满足下述三个条件的视图（视图名称为 ViewPractice5_1）。使用 product（商品）表作为参照表，假设表中包含初始状态的 8 行数据。

- 条件 1：销售单价大于等于 1000 日元。
- 条件 2：登记日期是 2009 年 9 月 20 日。
- 条件 3：包含商品名称、销售单价和登记日期三列。

对该视图执行 SELECT 语句的结果如下所示。

```
SELECT * FROM ViewPractice5_1;
```

执行结果

```
product_name | sale_price | regist_date
--------------+------------+------------
T 恤衫          |    1000    | 2009-09-20
菜刀            |    3000    | 2009-09-20
```

```
-- 创建视图的语句
CREATE VIEW ViewPractice5_1 AS
SELECT product_name, sale_price, regist_date
FROM product
WHERE sale_price >= 1000 AND regist_date = '2009-09-20';
```

练习 3.2

向习题一中创建的视图 ViewPractice5_1 中插入如下数据，会得到什么样的结果呢?

```
INSERT INTO ViewPractice5_1 VALUES ('刀子', 300, '2009-11-02');
```

```
mysql> INSERT INTO ViewPractice5_1 VALUES (' 刀子 ', 300, '2009-11-02');
ERROR 1423 (HY000): Field of view 'shop.viewpractice5_1' underlying table
doesn't have a default value
mysql> desc product;
+---------------+-------------+------+-----+---------+-------+
| Field         | Type        | Null | Key | Default | Extra |
+---------------+-------------+------+-----+---------+-------+
| product_id    | char(4)     | NO   | PRI | NULL    |       |
| product_name  | varchar(100)| NO   |     | NULL    |       |
| product_type  | varchar(32) | NO   |     | NULL    |       |
| sale_price    | int         | YES  |     | NULL    |       |
| purchase_price| int         | YES  |     | NULL    |       |
| regist_date   | date        | YES  |     | NULL    |       |
+---------------+-------------+------+-----+---------+-------+
6 rows in set (0.01 sec)
```

解析：插入时将会报错。
视图插入数据时，原表也会插入数据，而原表数据插入时不满足约束条件，所以会报错。（因为 `ViewPractice5_1` 的原表有三个带有 `NOT NULL` 约束的字段）

练习 3.3

请根据如下结果编写 SELECT 语句，其中 sale_price_all 列为全部商品的平均销售单价。

```
product_id | product_name | product_type | sale_price |
sale_price_all
------------+-------------+--------------+------------+--------------
-------
0001       | T 恤衫         | 衣服           | 1000       |
2097.5000000000000000
0002       | 打孔器         | 办公用品       | 500        |
2097.5000000000000000
0003       | 运动 T 恤       | 衣服           | 4000       |
2097.5000000000000000
0004       | 菜刀           | 厨房用具       | 3000       |
2097.5000000000000000
0005       | 高压锅         | 厨房用具       | 6800       |
2097.5000000000000000
0006       | 叉子           | 厨房用具       | 500        |
2097.5000000000000000
0007       | 擦菜板         | 厨房用具       | 880        |
2097.5000000000000000
0008       | 圆珠笔         | 办公用品       | 100        |
2097.5000000000000000
```

```sql
SELECT product_id,
       product_name,
       product_type,
       sale_price,
       (SELECT AVG(sale_price) FROM product) AS sale_price_all
FROM product;
```

练习 3.4

请根据习题一中的条件编写一条 SQL 语句，创建一幅包含如下数据的视图（名称为 AvgPriceByType）。

```
product_id | product_name | product_type | sale_price |
avg_sale_price
------------+-------------+--------------+------------+--------------
-------
0001       | T 恤衫         | 衣服          | 1000
|2500.0000000000000000
0002       | 打孔器         | 办公用品      | 500        |
300.0000000000000000
0003       | 运动 T 恤        | 衣服          | 4000
|2500.0000000000000000
0004       | 菜刀           | 厨房用具      | 3000
|2795.0000000000000000
0005       | 高压锅          | 厨房用具      | 6800
|2795.0000000000000000
0006       | 叉子           | 厨房用具      | 500
|2795.0000000000000000
0007       | 擦菜板          | 厨房用具      | 880
|2795.0000000000000000
0008       | 圆珠笔          | 办公用品      | 100        |
300.0000000000000000
```

提示：其中的关键是 avg_sale_price 列。与习题三不同，这里需要计算出的 是各商品种类的平均销售单价。这与使用关联子查询所得到的结果相同。 也就是说，该列可以使用关联子查询进行创建。问题就是应该在什么地方使用这个关联子查询。

```sql
-- 创建视图的语句
CREATE VIEW AvgPriceByType AS
SELECT product_id,
       product_name,
       product_type,
       sale_price,
       (SELECT AVG(sale_price)
         FROM product p2
```

```
          WHERE p1.product_type = p2.product_type
          GROUP BY p1.product_type) AS avg_sale_price
FROM product p1;

-- 确认视图内容
SELECT * FROM AvgPriceByType;

```

3.3 各种各样的函数

SQL 自带了各种各样的函数，极大提高了 SQL 语言的便利性。

所谓函数，类似一个黑盒子，你给它一个输入值，它便按照预设的程序定义给出返回值，输入值称为 `参数` 。

函数大致分为如下几类：

- 算术函数 （用来进行数值计算的函数）
- 字符串函数 （用来进行字符串操作的函数）
- 日期函数 （用来进行日期操作的函数）
- 转换函数 （用来转换数据类型和值的函数）
- 聚合函数 （用来进行数据聚合的函数）

函数总个数超过 200 个，不需要完全记住，常用函数有 30~50 个，其他不常用的函数使用时查阅文档即可。

3.3.1 算数函数

- `+ - * /` 四则运算在之前的章节介绍过，此处不再赘述。

为了演示其他的几个算数函数，在此构造 `samplemath` 表

```
-- DDL : 创建表
USE shop;
DROP TABLE IF EXISTS samplemath;
CREATE TABLE samplemath(
    m float(10,3),
    n INT,
    p INT
);

-- DML : 插入数据
START TRANSACTION; -- 开始事务
INSERT INTO samplemath(m, n, p) VALUES (500, 0, NULL);
```

```
INSERT INTO samplemath(m, n, p) VALUES (-180, 0, NULL);
INSERT INTO samplemath(m, n, p) VALUES (NULL, NULL, NULL);
INSERT INTO samplemath(m, n, p) VALUES (NULL, 7, 3);
INSERT INTO samplemath(m, n, p) VALUES (NULL, 5, 2);
INSERT INTO samplemath(m, n, p) VALUES (NULL, 4, NULL);
INSERT INTO samplemath(m, n, p) VALUES (8, NULL, 3);
INSERT INTO samplemath(m, n, p) VALUES (2.27, 1, NULL);
INSERT INTO samplemath(m, n, p) VALUES (5.555,2, NULL);
INSERT INTO samplemath(m, n, p) VALUES (NULL, 1, NULL);
INSERT INTO samplemath(m, n, p) VALUES (8.76, NULL, NULL);
COMMIT; -- 提交事务

-- 查询表内容
SELECT * FROM samplemath;
+----------+------+------+
| m        | n    | p    |
+----------+------+------+
|  500.000 |    0 | NULL |
| -180.000 |    0 | NULL |
|     NULL | NULL | NULL |
|     NULL |    7 |    3 |
|     NULL |    5 |    2 |
|     NULL |    4 | NULL |
|    8.000 | NULL |    3 |
|    2.270 |    1 | NULL |
|    5.555 |    2 | NULL |
|     NULL |    1 | NULL |
|    8.760 | NULL | NULL |
+----------+------+------+
11 rows in set (0.00 sec)
```

- ABS – 绝对值

  语法：`ABS(数值)`

  ABS 函数用于计算一个数字的绝对值，表示一个数到原点的距离。
  当 ABS 函数的参数为 `NULL` 时，返回值也是 `NULL`。

- MOD – 求余数

  语法：`MOD(被除数, 除数)`

  MOD 是计算除法余数（求余）的函数，是 modulo 的缩写。小数没有余数的概念，只能对整数列求余数。
  注意：主流的 DBMS 都支持 MOD 函数，只有 SQL Server 不支持该函数，其使用 `%` 符号来计算余数。

- ROUND – 四舍五入

  语法：`ROUND(对象数值，保留小数的位数)`

  ROUND 函数用来进行四舍五入操作。
  注意：当参数 **保留小数的位数** 为变量时，可能会遇到错误，请谨慎使用变量。

```
SELECT
m,
ABS(m) AS abs_col ,
n,
p,
MOD(n, p) AS mod_col,
ROUND(m,1) AS round_colS
FROM samplemath;
+----------+---------+------+------+---------+-----------+
| m        | abs_col | n    | p    | mod_col | round_col |
+----------+---------+------+------+---------+-----------+
|  500.000 | 500.000 |    0 | NULL |    NULL |     500.0 |
| -180.000 | 180.000 |    0 | NULL |    NULL |    -180.0 |
|     NULL |    NULL | NULL | NULL |    NULL |      NULL |
|     NULL |    NULL |    7 |    3 |       1 |      NULL |
|     NULL |    NULL |    5 |    2 |       1 |      NULL |
|     NULL |    NULL |    4 | NULL |    NULL |      NULL |
|    8.000 |   8.000 | NULL |    3 |    NULL |       8.0 |
|    2.270 |   2.270 |    1 | NULL |    NULL |       2.3 |
|    5.555 |   5.555 |    2 | NULL |    NULL |       5.6 |
|     NULL |    NULL |    1 | NULL |    NULL |      NULL |
|    8.760 |   8.760 | NULL | NULL |    NULL |       8.8 |
+----------+---------+------+------+---------+-----------+
11 rows in set (0.08 sec)
```

3.3.2 字符串函数

字符串函数也经常被使用，为了学习字符串函数，在此我们构造 `samplestr` 表。

```
-- DDL : 创建表
USE  shop;

DROP TABLE IF EXISTS samplestr;
CREATE TABLE samplestr(
    str1 VARCHAR (40),
    str2 VARCHAR (40),
    str3 VARCHAR (40)
);

```

```
-- DML: 插入数据
START TRANSACTION;
INSERT INTO samplestr (str1, str2, str3) VALUES ('opx', 'rt', NULL);
INSERT INTO samplestr (str1, str2, str3) VALUES ('abc', 'def', NULL);
INSERT INTO samplestr (str1, str2, str3) VALUES ('太阳', '月亮', '火
星');
INSERT INTO samplestr (str1, str2, str3) VALUES ('aaa', NULL, NULL);
INSERT INTO samplestr (str1, str2, str3) VALUES (NULL, 'xyz', NULL);
INSERT INTO samplestr (str1, str2, str3) VALUES ('@!#$%', NULL,
NULL);
INSERT INTO samplestr (str1, str2, str3) VALUES ('ABC', NULL, NULL);
INSERT INTO samplestr (str1, str2, str3) VALUES ('aBC', NULL, NULL);
INSERT INTO samplestr (str1, str2, str3) VALUES ('abc 哈哈',  'abc',
'ABC');
INSERT INTO samplestr (str1, str2, str3) VALUES ('abcdefabc', 'abc',
'ABC');
INSERT INTO samplestr (str1, str2, str3) VALUES ('micmic', 'i', 'I');
COMMIT;
-- 确认表中的内容
SELECT * FROM samplestr;
+-----------+------+------+
| str1      | str2 | str3 |
+-----------+------+------+
| opx       | rt   | NULL |
| abc       | def  | NULL |
| 太阳      | 月亮 | 火星 |
| aaa       | NULL | NULL |
| NULL      | xyz  | NULL |
| @!#$%     | NULL | NULL |
| ABC       | NULL | NULL |
| aBC       | NULL | NULL |
| abc 哈哈  | abc  | ABC  |
| abcdefabc | abc  | ABC  |
| micmic    | i    | I    |
+-----------+------+------+
11 rows in set (0.00 sec)
```

- CONCAT – 拼接

  语法：`CONCAT(str1, str2, str3)`

  MySQL 中使用 CONCAT 函数进行拼接。

- LENGTH – 字符串长度

  语法：`LENGTH(字符串)`

- LOWER – 小写转换

LOWER 函数只能针对英文字母使用，它会将参数中的字符串全都转换为小写。该函数不适用于英文字母以外的场合，不影响原本就是小写的字符。 类似的， UPPER 函数用于大写转换。

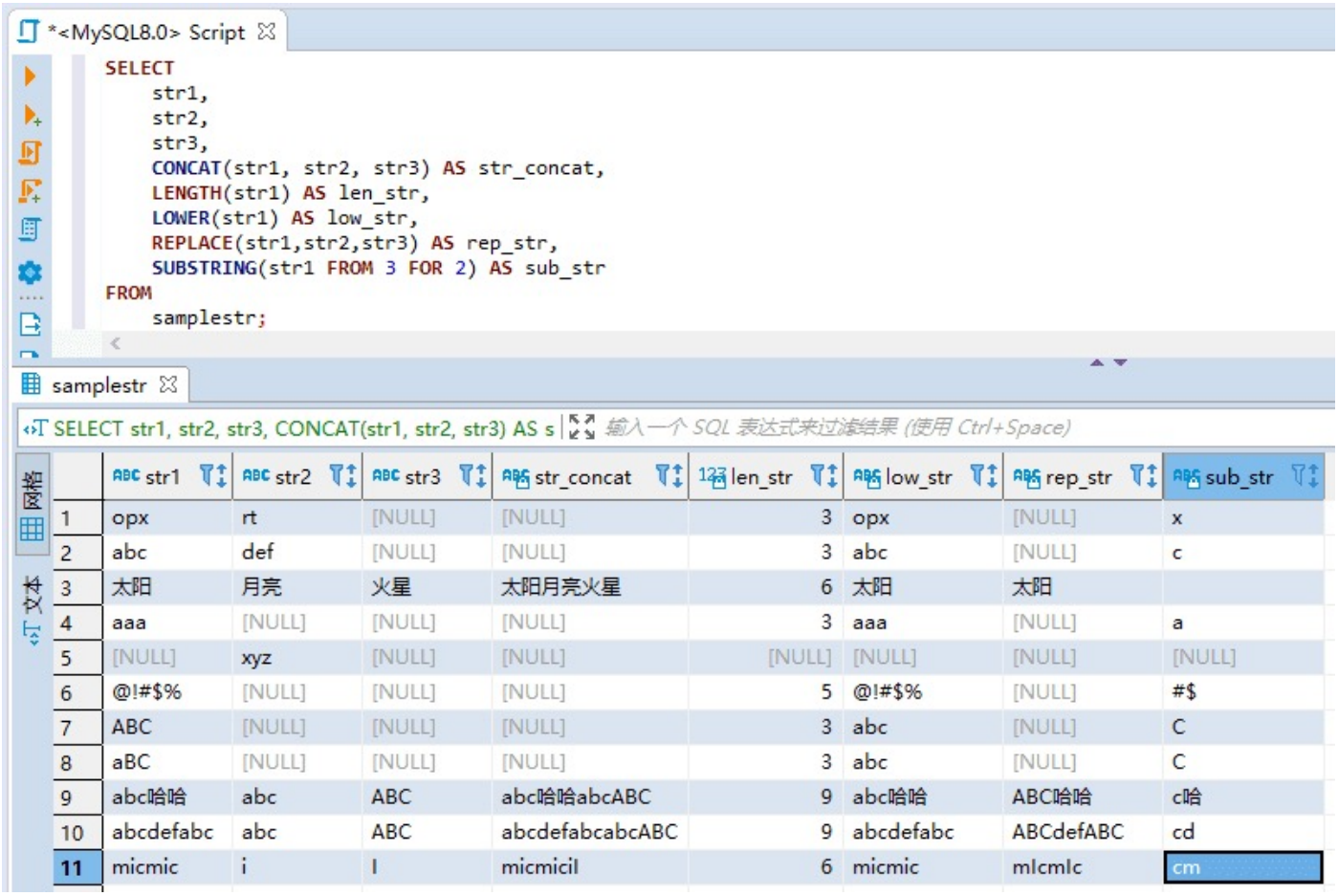
- REPLACE – 字符串的替换

  语法： `REPLACE(对象字符串，替换前的字符串，替换后的字符串)`

- SUBSTRING – 字符串的截取

  语法： `SUBSTRING （对象字符串 FROM 截取的起始位置 FOR 截取的字符数）`

  使用 SUBSTRING 函数 可以截取出字符串中的一部分字符串。截取的起始位置从字符串最左侧开始计算，索引值起始为 1。



	str1	str2	str3	str_concat	len_str	low_str	rep_str	sub_str
1	opx	rt	[NULL]	[NULL]	3	opx	[NULL]	x
2	abc	def	[NULL]	[NULL]	3	abc	[NULL]	c
3	太阳	月亮	火星	太阳月亮火星	6	太阳	太阳	
4	aaa	[NULL]	[NULL]	[NULL]	3	aaa	[NULL]	a
5	[NULL]	xyz	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]	[NULL]
6	@!#$%	[NULL]	[NULL]	[NULL]	5	@!#$%	[NULL]	#$
7	ABC	[NULL]	[NULL]	[NULL]	3	abc	[NULL]	C
8	aBC	[NULL]	[NULL]	[NULL]	3	abc	[NULL]	C
9	abc哈哈	abc	ABC	abc哈哈abcABC	9	abc哈哈	ABC哈哈	c哈
10	abcdefabc	abc	ABC	abcdefabcabcABC	9	abcdefabc	ABCdefABC	cd
11	micmic	i	I	micmicil	6	micmic	mIcmIc	cm

- **（扩展内容）SUBSTRING_INDEX – 字符串按索引截取**

  语法： `SUBSTRING_INDEX（原始字符串， 分隔符，n）`

  该函数用来获取原始字符串按照分隔符分割后，第 n 个分隔符之前（或之后）的子字符串，支持正向和反向索引，索引起始值分别为 1 和 -1。

```
SELECT SUBSTRING_INDEX('www.mysql.com', '.', 2);
+-----------------------------------------+
```

```
| SUBSTRING_INDEX('www.mysql.com', '.', 2) |
+------------------------------------------+
| www.mysql                                |
+------------------------------------------+
1 row in set (0.00 sec)
SELECT SUBSTRING_INDEX('www.mysql.com', '.', -2);
+-------------------------------------------+
| SUBSTRING_INDEX('www.mysql.com', '.', -2) |
+-------------------------------------------+
| mysql.com                                 |
+-------------------------------------------+
1 row in set (0.00 sec)
```

获取第 1 个元素比较容易，获取第 2 个元素 / 第 n 个元素可以采用二次拆分的写法。

```
SELECT SUBSTRING_INDEX('www.mysql.com', '.', 1);
+------------------------------------------+
| SUBSTRING_INDEX('www.mysql.com', '.', 1) |
+------------------------------------------+
| www                                      |
+------------------------------------------+
1 row in set (0.00 sec)
SELECT SUBSTRING_INDEX(SUBSTRING_INDEX('www.mysql.com', '.', 2), '.',
-1);
+-------------------------------------------------------------------
+
| SUBSTRING_INDEX(SUBSTRING_INDEX('www.mysql.com', '.', 2), '.', -1)
|
+-------------------------------------------------------------------
+
| mysql
|
+-------------------------------------------------------------------
+
1 row in set (0.00 sec)
```

3.3.3 日期函数

不同 DBMS 的日期函数语法各有不同，本课程介绍一些被标准 SQL 承认的可以应用于绝大多数 DBMS 的函数。特定 DBMS 的日期函数查阅文档即可。

- CURRENT_DATE – 获取当前日期

```
SELECT CURRENT_DATE;
+--------------+
| CURRENT_DATE |
```

```
+--------------+
| 2020-08-08   |
+--------------+
1 row in set (0.00 sec)
```

- CURRENT_TIME – 当前时间

```
SELECT CURRENT_TIME;
+--------------+
| CURRENT_TIME |
+--------------+
| 17:26:09     |
+--------------+
1 row in set (0.00 sec)
```

- CURRENT_TIMESTAMP – 当前日期和时间

```
SELECT CURRENT_TIMESTAMP;
+---------------------+
| CURRENT_TIMESTAMP   |
+---------------------+
| 2020-08-08 17:27:07 |
+---------------------+
1 row in set (0.00 sec)
```

- EXTRACT – 截取日期元素

  语法：`EXTRACT(日期元素 FROM 日期)`

  使用 EXTRACT 函数可以截取出日期数据中的一部分，例如“年”，“月”，或者“小时”“秒”等。该函数的返回值并不是日期类型而是数值类型

```
SELECT CURRENT_TIMESTAMP AS now,
EXTRACT(YEAR   FROM CURRENT_TIMESTAMP) AS year,
EXTRACT(MONTH  FROM CURRENT_TIMESTAMP) AS month,
EXTRACT(DAY    FROM CURRENT_TIMESTAMP) AS day,
EXTRACT(HOUR   FROM CURRENT_TIMESTAMP) AS hour,
EXTRACT(MINUTE FROM CURRENT_TIMESTAMP) AS MINute,
EXTRACT(SECOND FROM CURRENT_TIMESTAMP) AS second;
+---------------------+------+-------+------+------+--------+--------
+
| now                 | year | month | day  | hour | MINute | second
|
+---------------------+------+-------+------+------+--------+--------
+
| 2020-08-08 17:34:38 | 2020 |     8 |    8 |   17 |     34 |     38
```

```
|
+--------------------+------+-------+------+------+--------+--------
+
1 row in set (0.00 sec)
```

3.3.4 转换函数

“转换”这个词的含义非常广泛，在 SQL 中主要有两层意思：一是数据类型的转换，简称为类型转换，在英语中称为 `cast`；另一层意思是值的转换。

- CAST – 类型转换

  语法：`CAST（转换前的值 AS 想要转换的数据类型）`

```
-- 将字符串类型转换为数值类型
SELECT CAST('0001' AS SIGNED INTEGER) AS int_col;
+---------+
| int_col |
+---------+
|       1 |
+---------+
1 row in set (0.00 sec)
-- 将字符串类型转换为日期类型
SELECT CAST('2009-12-14' AS DATE) AS date_col;
+------------+
| date_col   |
+------------+
| 2009-12-14 |
+------------+
1 row in set (0.00 sec)
```

- COALESCE – 将 NULL 转换为其他值

  语法：`COALESCE(数据 1, 数据 2, 数据 3……)`

  COALESCE 是 SQL 特有的函数。该函数会返回可变参数 A 中左侧开始第 1 个不是 NULL 的值。参数个数是可变的，因此可以根据需要无限增加。

  在 SQL 语句中将 NULL 转换为其他值时就会用到转换函数。

```
SELECT
COALESCE(NULL, 11) AS col_1,
COALESCE(NULL, 'hello world', NULL) AS col_2,
COALESCE(NULL, NULL, '2020-11-01') AS col_3;
+-------+-------------+------------+
```

```
| col_1 | col_2       | col_3      |
+-------+-------------+------------+
|    11 | hello world | 2020-11-01 |
+-------+-------------+------------+
1 row in set (0.00 sec)
```

3.4 谓词

3.4.1 什么是谓词

谓词就是返回值为真值的函数。包括 `TRUE / FALSE / UNKNOWN` 。
谓词主要有以下几个：

- LIKE
- BETWEEN
- IS NULL、IS NOT NULL
- IN
- EXISTS

3.4.2 LIKE 谓词 – 用于字符串的部分一致查询

当需要进行字符串的部分一致查询时需要使用该谓词。部分一致大体可以分为前方一致、中间一致和后方一致三种类型。
首先我们来创建一张表

```
-- DDL ：创建表
CREATE TABLE samplelike(
    strcol VARCHAR(6) NOT NULL,
    PRIMARY KEY (strcol)
);

-- DML ：插入数据
START TRANSACTION; -- 开始事务
INSERT INTO samplelike (strcol) VALUES ('abcddd');
INSERT INTO samplelike (strcol) VALUES ('dddabc');
INSERT INTO samplelike (strcol) VALUES ('abdddc');
INSERT INTO samplelike (strcol) VALUES ('abcdd');
INSERT INTO samplelike (strcol) VALUES ('ddabc');
INSERT INTO samplelike (strcol) VALUES ('abddc');
COMMIT; -- 提交事务

```

```
SELECT * FROM samplelike;
+--------+
| strcol |
+--------+
| abcdd  |
| abcddd |
| abddc  |
| abdddc |
| ddabc  |
| dddabc |
+--------+
6 rows in set (0.00 sec)
```

- 前方一致：选取出“dddabc”

前方一致即作为查询条件的字符串（这里是“ddd”）与查询对象字符串起始部分相同。

```
SELECT *
FROM samplelike
WHERE strcol LIKE 'ddd%';
+--------+
| strcol |
+--------+
| dddabc |
+--------+
1 row in set (0.00 sec)
```

其中的 `%` 是代表“零个或多个任意字符串”的特殊符号，本例中代表“以 ddd 开头的所有字符串”。

- 中间一致：选取出“abcddd”,“dddabc”,“abdddc”

中间一致即查询对象字符串中含有作为查询条件的字符串，无论该字符串出现在对象字符串的最后还是中间都没有关系。

```
SELECT *
FROM samplelike
WHERE strcol LIKE '%ddd%';
+--------+
| strcol |
+--------+
| abcddd |
| abdddc |
| dddabc |
+--------+
3 rows in set (0.00 sec)
```

- 后方一致：选取出“abcddd“

后方一致即作为查询条件的字符串（这里是“ddd”）与查询对象字符串的末尾部分相同。

```
SELECT *
FROM samplelike
WHERE strcol LIKE '%ddd';
+--------+
| strcol |
+--------+
| abcddd |
+--------+
1 row in set (0.00 sec)
```

综合如上三种类型的查询可以看出，查询条件最宽松，也就是能够取得最多记录的是 `中间一致` 。这是因为它同时包含前方一致和后方一致的查询结果。

- `_` 下划线匹配任意 1 个字符

使用 _（下划线）来代替 %，与 % 不同的是，它代表了“任意 1 个字符”。

```
SELECT *
FROM samplelike
WHERE strcol LIKE 'abc__';
+--------+
| strcol |
+--------+
| abcdd  |
+--------+
1 row in set (0.00 sec)
```

3.4.3 BETWEEN 谓词 – 用于范围查询

使用 BETWEEN 可以进行范围查询。该谓词与其他谓词或者函数的不同之处在于它使用了 3 个参数。

```
-- 选取销售单价为 100～ 1000 元的商品
SELECT product_name, sale_price
FROM product
WHERE sale_price BETWEEN 100 AND 1000;
+--------------+------------+
| product_name | sale_price |
+--------------+------------+
| T 恤         |       1000 |
| 打孔器       |        500 |
```

```
| 叉子          |         500 |
| 擦菜板        |         880 |
| 圆珠笔        |         100 |
+--------------+-------------+
5 rows in set (0.00 sec)
```

BETWEEN 的特点就是结果中会包含 100 和 1000 这两个临界值，也就是 **闭区间**。如果不想让结果中包含临界值，那就必须使用 < 和 >。

```
SELECT product_name, sale_price
FROM product
WHERE sale_price > 100
AND sale_price < 1000;
+--------------+------------+
| product_name | sale_price |
+--------------+------------+
| 打孔器        |        500 |
| 叉子          |        500 |
| 擦菜板        |        880 |
+--------------+------------+
3 rows in set (0.00 sec)
```

3.4.4 IS NULL、 IS NOT NULL – 用于判断是否为 NULL

为了选取出某些值为 NULL 的列的数据，不能使用 =，而只能使用特定的谓词 IS NULL。

```
SELECT product_name, purchase_price
FROM product
WHERE purchase_price IS NULL;
+--------------+----------------+
| product_name | purchase_price |
+--------------+----------------+
| 叉子          |           NULL |
| 圆珠笔        |           NULL |
+--------------+----------------+
2 rows in set (0.00 sec)
```

与此相反，想要选取 NULL 以外的数据时，需要使用 `IS NOT NULL` 。

```
SELECT product_name, purchase_price
FROM product
WHERE purchase_price IS NOT NULL;
+--------------+----------------+
| product_name | purchase_price |
+--------------+----------------+
```

```
| T 恤          |            500 |
| 打孔器        |            320 |
| 运动 T 恤     |           2800 |
| 菜刀          |           2800 |
| 高压锅        |           5000 |
| 擦菜板        |            790 |
+--------------+----------------+
6 rows in set (0.00 sec)
```

3.4.5 IN 谓词 – OR 的简便用法

多个查询条件取并集时可以选择使用 `or` 语句。

```
-- 通过 OR 指定多个进货单价进行查询
SELECT product_name, purchase_price
FROM product
WHERE purchase_price = 320
OR purchase_price = 500
OR purchase_price = 5000;
+--------------+----------------+
| product_name | purchase_price |
+--------------+----------------+
| T 恤          |            500 |
| 打孔器        |            320 |
| 高压锅        |           5000 |
+--------------+----------------+
3 rows in set (0.00 sec)
```

虽然上述方法没有问题，但还是存在一点不足之处，那就是随着希望选取的对象越来越多，SQL 语句也会越来越长，阅读起来也会越来越困难。这时， 我们就可以使用 IN 谓词 `IN(值 1, 值 2, 值 3, ...) 来替换上述 SQL 语句。

```
SELECT product_name, purchase_price
FROM product
WHERE purchase_price IN (320, 500, 5000);
+--------------+----------------+
| product_name | purchase_price |
+--------------+----------------+
| T 恤          |            500 |
| 打孔器        |            320 |
| 高压锅        |           5000 |
+--------------+----------------+
3 rows in set (0.00 sec)
```

上述语句简洁了很多，可读性大幅提高。

反之，希望选取出“进货单价不是 320 元、 500 元、 5000 元”的商品时，可以使用否定形式 NOT IN 来实现。

```
SELECT product_name, purchase_price
FROM product
WHERE purchase_price NOT IN (320, 500, 5000);
+--------------+----------------+
| product_name | purchase_price |
+--------------+----------------+
| 运动 T 恤       |           2800 |
| 菜刀           |           2800 |
| 擦菜板         |            790 |
+--------------+----------------+
3 rows in set (0.00 sec)
```

需要注意的是，在使用 IN 和 NOT IN 时是无法选取出 NULL 数据的。
实际结果也是如此，上述两组结果中都不包含进货单价为 NULL 的叉子和圆珠笔。 NULL 只能使用 IS NULL 和 IS NOT NULL 来进行判断。

3.4.6 使用子查询作为 IN 谓词的参数

- IN 和子查询

IN 谓词（NOT IN 谓词）具有其他谓词所没有的用法，那就是可以使用子查询作为其参数。我们已经在 5-2 节中学习过了，子查询就是 SQL 内部生成的表，因此也可以说“能够将表作为 IN 的参数”。同理，我们还可以说“能够将视图作为 IN 的参数”。

在此，我们创建一张新表 `shopproduct` 显示出哪些商店销售哪些商品。

```
-- DDL : 创建表
DROP TABLE IF EXISTS shopproduct;

CREATE TABLE shopproduct(
    shop_id CHAR(4)     NOT NULL,
    shop_name VARCHAR(200) NOT NULL,
    product_id CHAR(4)      NOT NULL,
    quantity INTEGER      NOT NULL,
    PRIMARY KEY (shop_id, product_id) -- 指定主键
);

-- DML : 插入数据
START TRANSACTION; -- 开始事务
INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity)
VALUES ('000A', '东京', '0001', 30);
INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity)
```

```
VALUES ('000A', '东京', '0002', 50);
INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity)
VALUES ('000A', '东京', '0003', 15);
INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity)
VALUES ('000B', '名古屋', '0002', 30);
INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity)
VALUES ('000B', '名古屋', '0003', 120);
INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity)
VALUES ('000B', '名古屋', '0004', 20);
INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity)
VALUES ('000B', '名古屋', '0006', 10);
INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity)
VALUES ('000B', '名古屋', '0007', 40);
INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity)
VALUES ('000C', '大阪', '0003', 20);
INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity)
VALUES ('000C', '大阪', '0004', 50);
INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity)
VALUES ('000C', '大阪', '0006', 90);
INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity)
VALUES ('000C', '大阪', '0007', 70);
INSERT INTO shopproduct (shop_id, shop_name, product_id, quantity)
VALUES ('000D', '福冈', '0001', 100);
COMMIT; -- 提交事务
SELECT * FROM shopproduct;
+---------+-----------+------------+----------+
| 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 |
+---------+-----------+------------+----------+
13 rows in set (0.00 sec)
```

由于单独使用商店编号（shop_id）或者商品编号（product_id）不能区分表中每一行数据，因此指定了 2 列作为主键（primary key）对商店和商品进行组合，用来唯一确定每一行数据。

假设我么需要取出大阪在售商品的销售单价，该如何实现呢?

第一步，取出大阪门店的在售商品 `product_id` ;

第二步，取出大阪门店在售商品的销售单价 `sale_price`

```
-- step1: 取出大阪门店的在售商品 `product_id`
SELECT product_id
FROM shopproduct
WHERE shop_id = '000C';
+------------+
| product_id |
+------------+
| 0003       |
| 0004       |
| 0006       |
| 0007       |
+------------+
4 rows in set (0.00 sec)
```

上述语句取出了大阪门店的在售商品编号，接下来，我么可以使用上述语句作为第二步的查询条件来使用了。

```
-- step2: 取出大阪门店在售商品的销售单价 `sale_price`
SELECT product_name, sale_price
FROM product
WHERE product_id IN (SELECT product_id
                     FROM shopproduct
                     WHERE shop_id = '000C');
+--------------+------------+
| product_name | sale_price |
+--------------+------------+
| 运动 T 恤      |       4000 |
| 菜刀           |       3000 |
| 叉子           |        500 |
| 擦菜板         |        880 |
+--------------+------------+
4 rows in set (0.00 sec)
```

根据第 5 章学习的知识，子查询是从最内层开始执行的（由内而外），因此，上述语句的子查询执行之后，sql 展开成下面的语句

```
-- 子查询展开后的结果
SELECT product_name, sale_price
FROM product
WHERE product_id IN ('0003', '0004', '0006', '0007');
```

```
+--------------+------------+
| product_name | sale_price |
+--------------+------------+
| 运动 T 恤       |       4000 |
| 菜刀           |       3000 |
| 叉子           |        500 |
| 擦菜板         |        880 |
+--------------+------------+
4 rows in set (0.00 sec)
```

可以看到，子查询转换之后变为 in 谓词用法，你理解了吗？
或者，你会疑惑既然 in 谓词也能实现，那为什么还要使用子查询呢？这里给出两点原因：

1. 实际生活中，某个门店的在售商品是不断变化的，使用 in 谓词就需要经常更新 sql 语句，降低了效率，提高了维护成本；
2. 实际上，某个门店的在售商品可能有成百上千个，手工维护在售商品编号真是个大工程。

使用子查询即可保持 sql 语句不变，极大提高了程序的可维护性，这是系统开发中需要重点考虑的内容。

- NOT IN 和子查询

NOT IN 同样支持子查询作为参数，用法和 in 完全一样。

```sql
-- NOT IN 使用子查询作为参数，取出未在大阪门店销售的商品的销售单价
SELECT product_name, sale_price
FROM product
WHERE product_id NOT IN (SELECT product_id
                         FROM shopproduct
                         WHERE shop_id = '000A');
+--------------+------------+
| product_name | sale_price |
+--------------+------------+
| 菜刀           |       3000 |
| 高压锅         |       6800 |
| 叉子           |        500 |
| 擦菜板         |        880 |
| 圆珠笔         |        100 |
+--------------+------------+
5 rows in set (0.00 sec)
```

3.4.7 EXIST 谓词

EXIST 谓词的用法理解起来有些难度。

1. EXIST 的使用方法与之前的都不相同
2. 语法理解起来比较困难
3. 实际上即使不使用 EXIST，基本上也都可以使用 IN（或者 NOT IN）来代替

这么说的话，还有学习 EXIST 谓词的必要吗？答案是肯定的，因为一旦能够熟练使用 EXIST 谓词，就能体会到它极大的便利性。

不过，你不用过于担心，本课程介绍一些基本用法，日后学习时可以多多留意 EXIST 谓词的用法，以期能够在达到 SQL 中级水平时掌握此用法。

- EXIST 谓词的使用方法

谓词的作用就是**“判断是否存在满足某种条件的记录”**。

如果存在这样的记录就返回真（TRUE），如果不存在就返回假（FALSE）。

EXIST（存在）谓词的主语是“记录”。

我们继续以 IN 和子查询 中的示例，使用 EXIST 选取出大阪门店在售商品的销售单价。

```
SELECT product_name, sale_price
FROM product AS p
WHERE EXISTS (SELECT *
              FROM shopproduct AS sp
              WHERE sp.shop_id = '000C'
                  AND sp.product_id = p.product_id);
+--------------+------------+
| product_name | sale_price |
+--------------+------------+
| 运动 T 恤      |       4000 |
| 菜刀          |       3000 |
| 叉子          |        500 |
| 擦菜板         |        880 |
+--------------+------------+
4 rows in set (0.00 sec)
```

- EXIST 的参数

之前我们学过的谓词，基本上都是像“列 LIKE 字符串”或者“ 列 BETWEEN 值 1 AND 值 2”这样需要指定 2 个以上的参数，而 EXIST 的左侧并没有任何参数。因为 EXIST 是只有 1 个参数的谓词。 所以，EXIST 只需要在右侧书写 1 个参数，该参数通常都会是一个子查询。

```
(SELECT *
 FROM shopproduct AS sp
 WHERE sp.shop_id = '000C'
    AND sp.product_id = p.product_id)
```

上面这样的子查询就是唯一的参数。确切地说，由于通过条件“SP.product_id = P.product_id”将 product 表和 shopproduct 表进行了联接，因此作为参数的是关联子查询。 EXIST 通常会使用关联子查询作为参数。

- 子查询中的 SELECT

由于 EXIST 只关心记录是否存在，因此返回哪些列都没有关系。 EXIST 只会判断是否存在满足子查询中 WHERE 子句指定的条件“商店编号（shop_id）为 '000C'，商品（product）表和商店

商品（shopproduct）表中商品编号（product_id）相同”的记录，只有存在这样的记录时才返回真（TRUE）。

因此，使用下面的查询语句，查询结果也不会发生变化。

```
SELECT product_name, sale_price
FROM product AS p
WHERE EXISTS (SELECT 1 -- 这里可以书写适当的常数
              FROM shopproduct AS sp
              WHERE sp.shop_id = '000C'
                  AND sp.product_id = p.product_id);
+--------------+------------+
| product_name | sale_price |
+--------------+------------+
| 运动 T 恤       |       4000 |
| 菜刀           |       3000 |
| 叉子           |        500 |
| 擦菜板          |        880 |
+--------------+------------+
4 rows in set (0.00 sec)
```

> 大家可以把在 EXIST 的子查询中书写 SELECT * 当作 SQL 的一种习惯。

- 使用 NOT EXIST 替换 NOT IN

就像 EXIST 可以用来替换 IN 一样， NOT IN 也可以用 NOT EXIST 来替换。

下面的代码示例取出，不在大阪门店销售的商品的销售单价。

```
SELECT product_name, sale_price
```

```
FROM product AS p
WHERE NOT EXISTS (SELECT *
                  FROM shopproduct AS sp
                  WHERE sp.shop_id = '000A'
                      AND sp.product_id = p.product_id);
+--------------+------------+
| product_name | sale_price |
+--------------+------------+
| 菜刀         |       3000 |
| 高压锅       |       6800 |
| 叉子         |        500 |
| 擦菜板       |        880 |
| 圆珠笔       |        100 |
+--------------+------------+
5 rows in set (0.00 sec)
```

NOT EXIST 与 EXIST 相反，当“不存在”满足子查询中指定条件的记录时返回真（TRUE）。

3.5 CASE 表达式

3.5.1 什么是 CASE 表达式?

CASE 表达式是函数的一种，是 SQL 中数一数二的重要功能，有必要好好学习一下。

CASE 表达式是在区分情况时使用的，这种情况的区分在编程中通常称为（条件）分支。

CASE 表达式的语法分为简单 CASE 表达式和搜索 CASE 表达式两种。由于搜索 CASE 表达式包含简单 CASE 表达式的全部功能。本课程将重点介绍搜索 CASE 表达式。

语法：

```
CASE WHEN < 求值表达式 > THEN < 表达式 >
     WHEN < 求值表达式 > THEN < 表达式 >
     WHEN < 求值表达式 > THEN < 表达式 >
     .
     .
     .
ELSE < 表达式 >
END
```

上述语句执行时，依次判断 when 表达式是否为真值，是则执行 THEN 后的语句，如果所有的 when 表达式均为假，则执行 ELSE 后的语句。
无论多么庞大的 CASE 表达式，最后也只会返回一个值。

3.5.2 CASE 表达式的使用方法

假设现在 要实现如下结果：

```
A ： 衣服
B ： 办公用品
C ： 厨房用具
```

因为表中的记录并不包含“A： ”或者“B： ”这样的字符串，所以需要在 SQL 中进行添加。并将“A： ”“B： ”“C： ”与记录结合起来。

- **应用场景 1：根据不同分支得到不同列值**

```
SELECT  product_name,
        CASE
            WHEN product_type = '衣服'      THEN CONCAT('A
: ',product_type)
            WHEN product_type = '办公用品'  THEN CONCAT('B
: ',product_type)
            WHEN product_type = '厨房用具'  THEN CONCAT('C
: ',product_type)
            ELSE NULL
        END

        AS abc_product_type
FROM  product;
+--------------+------------------+
| product_name | abc_product_type |
+--------------+------------------+
| T 恤           | A ：  衣服         |
| 打孔器         | B ：  办公用品     |
| 运动 T 恤       | A ：  衣服         |
| 菜刀           | C ：  厨房用具     |
| 高压锅         | C ：  厨房用具     |
| 叉子           | C ：  厨房用具     |
| 擦菜板         | C ：  厨房用具     |
| 圆珠笔         | B ：  办公用品     |
+--------------+------------------+
8 rows in set (0.00 sec)
```

ELSE 子句也可以省略不写，这时会被默认为 ELSE NULL。但为了防止有人漏读，还是希望大家能够显示地写出 ELSE 子句。
此外， CASE 表达式最后的“END”是不能省略的，请大家特别注意不要遗漏。忘记书写 END 会发生语法错误，这也是初学时最容易犯的错误。

- **应用场景 2：实现列方向上的聚合**

通常我们使用如下代码实现行的方向上不同种类的聚合（这里是 sum）

```
SELECT product_type,
       SUM(sale_price) AS sum_price
FROM product
GROUP BY product_type;
+--------------+-----------+
| product_type | sum_price |
+--------------+-----------+
| 衣服         |      5000 |
| 办公用品     |       600 |
| 厨房用具     |     11180 |
+--------------+-----------+
3 rows in set (0.00 sec)
```

假如要在列的方向上展示不同种类额聚合值，该如何写呢?

```
sum_price_clothes | sum_price_kitchen | sum_price_office
------------------+-------------------+-----------------
             5000 |             11180 |             600
```

聚合函数 + CASE WHEN 表达式即可实现该效果

```
-- 对按照商品种类计算出的销售单价合计值进行行列转换
SELECT
        SUM(CASE WHEN product_type = '衣服' THEN sale_price ELSE 0
END)
        AS sum_price_clothes,
        SUM(CASE WHEN product_type = '厨房用具' THEN sale_price ELSE 0
END)
        AS sum_price_kitchen,
        SUM(CASE WHEN product_type = '办公用品' THEN sale_price ELSE 0
END)
        AS sum_price_office
FROM product;
+------------------+-------------------+------------------+
| sum_price_clothes | sum_price_kitchen | sum_price_office |
+------------------+-------------------+------------------+
|              5000 |             11180 |              600 |
+------------------+-------------------+------------------+
1 row in set (0.00 sec)
```

- **（扩展内容）应用场景 3：实现行转列**

假设有如下图表的结构

name	subject	score
张三	语文	93
张三	数学	88
张三	外语	91
李四	语文	87
李四	数学	90
李四	外语	77

计划得到如下的图表结构

name	chinese	math	english
张三	93	88	91
李四	87	90	77

聚合函数 + CASE WHEN 表达式即可实现该转换

```
-- CASE WHEN 实现数字列 score 行转列
SELECT name,
       SUM(CASE WHEN subject = '语文' THEN score ELSE null END) as
chinese,
       SUM(CASE WHEN subject = '数学' THEN score ELSE null END) as
math,
       SUM(CASE WHEN subject = '外语' THEN score ELSE null END) as
english
  FROM score
 GROUP BY name;
+------+---------+------+---------+
| name | chinese | math | english |
+------+---------+------+---------+
| 张三 |      93 |   88 |      91 |
| 李四 |      87 |   90 |      77 |
+------+---------+------+---------+
2 rows in set (0.00 sec)
```

上述代码实现了数字列 score 的行转列，也可以实现文本列 subject 的行转列

```
-- CASE WHEN 实现文本列 subject 行转列
SELECT name,
       MAX(CASE WHEN subject = '语文' THEN subject ELSE null END) as
chinese,
       MAX(CASE WHEN subject = '数学' THEN subject ELSE null END) as
math,
       MIN(CASE WHEN subject = '外语' THEN subject ELSE null END) as
english
FROM score
GROUP BY name;
+------+---------+------+---------+
```

```
| name | chinese | math | english |
+------+---------+------+---------+
| 张三 | 语文    | 数学 | 外语    |
| 李四 | 语文    | 数学 | 外语    |
+------+---------+------+---------+
2 rows in set (0.00 sec
```

总结：

- **当待转换列为数字时，可以使用 `SUM AVG MAX MIN` 等聚合函数；**
- **当待转换列为文本时，可以使用 `MAX MIN` 等聚合函数**

练习题 - 第二部分

练习 3.5

运算或者函数中含有 NULL 时，结果全都会变为 NULL？（判断题）

正确

练习 3.6

对本章中使用的 product（商品）表执行如下 2 条 SELECT 语句，能够得到什么样的结果呢?

1. 第一条

```sql
SELECT product_name, purchase_price
FROM product
WHERE purchase_price NOT IN (500, 2800, 5000);
```

2. 第二条

```sql
SELECT product_name, purchase_price
FROM product
WHERE purchase_price NOT IN (500, 2800, 5000, NULL);
```

练习 3.7

按照销售单价（ sale_price）对练习 6.1 中的 product（商品）表中的商品进行如下分类。

- 低档商品：销售单价在 1000 日元以下（T 恤衫、办公用品、叉子、擦菜板、 圆珠笔）

- 中档商品：销售单价在 1001 日元以上 3000 日元以下（菜刀）
- 高档商品：销售单价在 3001 日元以上（运动 T 恤、高压锅）

请编写出统计上述商品种类中所包含的商品数量的 SELECT 语句，结果如下所示。

执行结果

```
low_price | mid_price | high_price
----------+-----------+------------
        5 |         1 |          2
```

```
SELECT
SUM(CASE WHEN sale_price <= 1000 THEN 1 ELSE 0 END) AS low_price,

SUM(CASE WHEN sale_price BETWEEN 1001 AND 3000 THEN 1 ELSE 0 END) AS
mid_price,

SUM(CASE WHEN sale_price >= 3001 THEN 1 ELSE 0 END) AS high_price

FROM product;
```