🤔👀SQL语言 - SQL语法基础

基础

模式定义了数据如何存储、存储什么样的数据以及数据如何分解等信息，数据库和表都有模式。

主键的值不允许修改，也不允许复用(不能使用已经删除的主键值赋给新数据行的主键)。

SQL(Structured Query Language)，标准 SQL 由 ANSI 标准委员会管理，从而称为 ANSI SQL。各个 DBMS 都有自己的实现，如 PL/SQL、Transact-SQL 等。

SQL 语句不区分大小写，但是数据库表名、列名和值是否区分依赖于具体的 DBMS 以及配置。

SQL 支持以下三种注释：

```
# 注释
SELECT *
FROM mytable; -- 注释
/* 注释1
   注释2 */
```

数据库创建与使用：

```
CREATE DATABASE test;
USE test;
```

创建表

```
CREATE TABLE mytable (
  id INT NOT NULL AUTO_INCREMENT,
  col1 INT NOT NULL DEFAULT 1,
  col2 VARCHAR(45) NULL,
  col3 DATE NULL,
  PRIMARY KEY (`id`));
```

修改表

添加列

```
ALTER TABLE mytable
ADD col CHAR(20);
```

删除列

```
ALTER TABLE mytable
DROP COLUMN col;
```

删除表

```
DROP TABLE mytable;
```

插入

普通插入

```
INSERT INTO mytable(col1, col2)
VALUES(val1, val2);
```

插入检索出来的数据

```
INSERT INTO mytable1(col1, col2)
SELECT col1, col2
FROM mytable2;
```

将一个表的内容插入到一个新表

```
CREATE TABLE newtable AS
SELECT * FROM mytable;
```

删除

```
DELETE FROM mytable
WHERE id = 1;
```

TRUNCATE TABLE 可以清空表，也就是删除所有行。

```
TRUNCATE TABLE mytable;
```

使用更新和删除操作时一定要用 `WHERE` 子句，不然会把整张表的数据都破坏。可以先用 `SELECT` 语句进行测试，防止错误删除。

查询

DISTINCT

相同值只会出现一次。它作用于所有列，也就是说所有列的值都相同才算相同。

```
SELECT DISTINCT col1, col2
FROM mytable;
```

LIMIT

限制返回的行数。可以有两个参数，第一个参数为起始行，从 `0` 开始；第二个参数为返回的总行数。

返回前 `5` 行：

```
SELECT *
FROM mytable
LIMIT 5;
```

```
SELECT *
FROM mytable
LIMIT 0, 5;
```

返回第 3 ~ 5 行:

```
SELECT *
FROM mytable
LIMIT 2, 3;
```

排序

- **ASC** : 升序(默认)
- **DESC** : 降序

可以按多个列进行排序，并且为每个列指定不同的排序方式:

```
SELECT *
FROM mytable
ORDER BY col1 DESC, col2 ASC;
```

过滤

不进行过滤的数据非常大，导致通过网络传输了多余的数据，从而浪费了网络带宽。因此尽量使用 SQL 语句来过滤不必要的数据，而不是传输所有的数据到客户端中然后由客户端进行过滤。

```
SELECT *
FROM mytable
WHERE col IS NULL;
```

下表显示了 WHERE 子句可用的操作符

操作符	说明
=	等于
<	小于
>	大于
<> !=	不等于
<= !>	小于等于
<= !>	大于等于
BETWEEN	在两个值之间
IS NULL	为NULL值

应该注意到，NULL 与 0、空字符串都不同。

AND 和 OR 用于连接多个过滤条件。优先处理 AND，当一个过滤表达式涉及到多个 AND 和 OR 时，可以使用 () 来决定优先级，使得优先级关系更清晰。

IN 操作符用于匹配一组值，其后也可以接一个 SELECT 子句，从而匹配子查询得到的一组值。

NOT 操作符用于否定一个条件。

通配符

通配符也是用在过滤语句中，但它只能用于文本字段。

- **%** 匹配 >=0 个任意字符；
- **_** 匹配 ==1 个任意字符；
- **[ ]** 可以匹配集合内的字符，例如 [ab] 将匹配字符 a 或者 b。用脱字符 ^ 可以对其进行否定，也就是不匹配集合内的字符。

使用 Like 来进行通配符匹配。

```
SELECT *
FROM mytable
WHERE col LIKE '[^AB]%'; -- 不以 A 和 B 开头的任意文本
```

不要滥用通配符，通配符位于开头处匹配会非常慢。

计算字段

在数据库服务器上完成数据的转换和格式化的工作往往比客户端上快得多，并且转换和格式化后的数据量更少的话可以减少网络通信量。

计算字段通常需要使用 **AS** 来取别名，否则输出的时候字段名为计算表达式。

```
SELECT col1 * col2 AS alias
FROM mytable;
```

CONCAT() 用于连接两个字段。许多数据库会使用空格把一个值填充为列宽，因此连接的结果会出现一些不必要的空格，使用 **TRIM()** 可以去除首尾空格。

```
SELECT CONCAT(TRIM(col1), '(', TRIM(col2), ')') AS concat_col
FROM mytable;
```

函数

各个 DBMS 的函数都是不相同的，因此不可移植，以下主要是 MySQL 的函数。

汇总

函数	说明
AVG()	返回某列的平均值
COUNT()	返回某列的行数
MAX()	返回某列的最大值
MIN()	返回某列的最小值
SUM()	返回某列值之和

AVG() 会忽略 NULL 行。

使用 DISTINCT 可以让汇总函数值汇总不同的值。

```
SELECT AVG(DISTINCT col1) AS avg_col
FROM mytable;
```

文本处理

函数	说明
LEFT()	左边的字符
RIGHT()	右边的字符
LOWER()	转换为小写字符
UPPER()	转换为大写字符
LTIRM()	去除左边的空格
RTRIM()	去除右边的空格
LENGTH()	长度
SOUNDEX()	转换为语音值

其中， **SOUNDEX()** 可以将一个字符串转换为描述其语音表示的字母数字模式。

```
SELECT *
FROM mytable
WHERE SOUNDEX(col1) = SOUNDEX('apple')
```

日期和时间处理

- 日期格式：YYYY-MM-DD
- 时间格式：HH:MM:SS

函数	说明
AddDate()	增加一个日期(天，周等)
AddTime()	增加一个时间(时、分等)
CurDate()	返回当前日期
CurTime()	返回当前时间
Date()	返回日期时间的日期部分
DateDiff()	计算两个日期之差
Date_Add()	高度灵活的日期运算函数
Date_Format()	返回一个格式化的日期或时间串
Day()	返回一个日期的天数部分
DayOfWeek()	对于一个日期，返回对应的星期几
Hour()	返回一个时间的小时部分
Minute()	返回一个时间的分钟部分
Month()	返回一个日期的月份部分
Now()	返回当前日期和时间
Second()	返回一个时间的秒部分
Time()	返回一个日期时间的时间部分
Year()	返回一个日期的年份部分

查询:

```
mysql> SELECT NOW();
```

结果:

```
2018-4-14 20:25:11
```

数值处理

函数	说明
SIN()	正弦
COS()	余弦
TAN()	正切
ABS()	绝对值
SQRT()	平方根
MOD()	余数
EXP()	指数
PI()	圆周率
RAND()	随机数

分组

分组就是把具有相同的数据值的行放在同一组中。

可以对同一分组数据使用汇总函数进行处理，例如求分组数据的平均值等。

指定的分组字段除了能按该字段进行分组，也会自动按该字段进行排序。

```
SELECT col, COUNT(*) AS num
FROM mytable
GROUP BY col;
```

GROUP BY 自动按分组字段进行排序，ORDER BY 也可以按汇总字段来进行排序。

```
SELECT col, COUNT(*) AS num
FROM mytable
GROUP BY col
ORDER BY num;
```

WHERE 过滤行，HAVING 过滤分组，行过滤应当先于分组过滤。

```
SELECT col, COUNT(*) AS num
FROM mytable
WHERE col > 2
GROUP BY col
HAVING num >= 2;
```

分组规定：

- GROUP BY 子句出现在 WHERE 子句之后，ORDER BY 子句之前；
- 除了汇总字段外，SELECT 语句中的每一字段都必须在 GROUP BY 子句中给出；
- NULL 的行会单独分为一组；
- 大多数 SQL 实现不支持 GROUP BY 列具有可变长度的数据类型。

子查询

子查询中只能返回一个字段的数据。

可以将子查询的结果作为 WHRER 语句的过滤条件：

```
SELECT *
FROM mytable1
WHERE col1 IN (SELECT col2
               FROM mytable2);
```

下面的语句可以检索出客户的订单数量，子查询语句会对第一个查询检索出的每个客户执行一次：

```
SELECT cust_name, (SELECT COUNT(*)
                   FROM Orders
                   WHERE Orders.cust_id = Customers.cust_id)
                   AS orders_num
FROM Customers
ORDER BY cust_name;
```

连接

连接用于连接多个表，使用 JOIN 关键字，并且条件语句使用 ON 而不是 WHERE。

连接可以替换子查询，并且比子查询的效率一般会更快。

可以用 AS 给列名、计算字段和表名取别名，给表名取别名是为了简化 SQL 语句以及连接相同表。

内连接

内连接又称等值连接，使用 INNER JOIN 关键字。

```
SELECT A.value, B.value
FROM tablea AS A INNER JOIN tableb AS B
ON A.key = B.key;
```

可以不明确使用 INNER JOIN，而使用普通查询并在 WHERE 中将两个表中要连接的列用等值方法连接起来。

```
SELECT A.value, B.value
FROM tablea AS A, tableb AS B
WHERE A.key = B.key;
```

在没有条件语句的情况下返回笛卡尔积。

自连接

自连接可以看成内连接的一种，只是连接的表是自身而已。

一张员工表，包含员工姓名和员工所属部门，要找出与 Jim 处在同一部门的所有员工姓名。

子查询版本

```
SELECT name
FROM employee
WHERE department = (
      SELECT department
      FROM employee
      WHERE name = "Jim");
```

自连接版本

```
SELECT e1.name
FROM employee AS e1 INNER JOIN employee AS e2
ON e1.department = e2.department
      AND e2.name = "Jim";
```

自然连接

自然连接是把同名列通过等值测试连接起来的，同名列可以有多个。

内连接和自然连接的区别：内连接提供连接的列，而自然连接自动连接所有同名列。

```
SELECT A.value, B.value
FROM tablea AS A NATURAL JOIN tableb AS B;
```

外连接

外连接保留了没有关联的那些行。分为左外连接，右外连接以及全外连接，左外连接就是保留左表没有关联的行。

检索所有顾客的订单信息，包括还没有订单信息的顾客。

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers LEFT OUTER JOIN Orders
ON Customers.cust_id = Orders.cust_id;
```

customers 表：

cust_id	cust_name
1	a
2	b
3	c

orders 表：

order_id	cust_id
1	1
2	1
3	3
4	3

结果：

cust_id	cust_name	order_id
1	a	1
1	a	2
3	c	3
3	c	4
2	b	Null

组合查询

使用 **UNION** 来组合两个查询，如果第一个查询返回 M 行，第二个查询返回 N 行，那么组合查询的结果一般为 M+N 行。

每个查询必须包含相同的列、表达式和聚集函数。

默认会去除相同行，如果需要保留相同行，使用 UNION ALL。

只能包含一个 ORDER BY 子句，并且必须位于语句的最后。

```
SELECT col
FROM mytable
WHERE col = 1
UNION
SELECT col
FROM mytable
WHERE col =2;
```

视图

视图是虚拟的表，本身不包含数据，也就不能对其进行索引操作。

对视图的操作和对普通表的操作一样。

视图具有如下好处：

- 简化复杂的 SQL 操作，比如复杂的连接；
- 只使用实际表的一部分数据；
- 通过只给用户访问视图的权限，保证数据的安全性；
- 更改数据格式和表示。

```
CREATE VIEW myview AS
SELECT Concat(col1, col2) AS concat_col, col3*col4 AS compute_col
FROM mytable
WHERE col5 = val;
```

存储过程

存储过程可以看成是对一系列 SQL 操作的批处理。

使用存储过程的好处：

- 代码封装，保证了一定的安全性；
- 代码复用；
- 由于是预先编译，因此具有很高的性能。

命令行中创建存储过程需要自定义分隔符，因为命令行是以 ; 为结束符，而存储过程中也包含了分号，因此会错误把这部分分号当成是结束符，造成语法错误。

包含 in、out 和 inout 三种参数。

给变量赋值都需要用 select into 语句。

每次只能给一个变量赋值，不支持集合的操作。

```
delimiter //

create procedure myprocedure( out ret int )
    begin
        declare y int;
        select sum(col1)
        from mytable
        into y;
        select y*y into ret;
    end //

delimiter ;
```

```
call myprocedure(@ret);
select @ret;
```

游标

在存储过程中使用游标可以对一个结果集进行移动遍历。

游标主要用于交互式应用，其中用户需要对数据集中的任意行进行浏览和修改。

使用游标的四个步骤：

1. 声明游标，这个过程没有实际检索出数据；
2. 打开游标；
3. 取出数据；
4. 关闭游标；

```
delimiter //
create procedure myprocedure(out ret int)
    begin
        declare done boolean default 0;

        declare mycursor cursor for
        select col1 from mytable;
 # 定义了一个 continue handler，当 sqlstate '02000' 这个条件出现时，会执行 set done = 1
        declare continue handler for sqlstate '02000' set done = 1;

        open mycursor;

        repeat
            fetch mycursor into ret;
            select ret;
        until done end repeat;

        close mycursor;
    end //
```

```
delimiter ;
```

触发器

触发器会在某个表执行以下语句时而自动执行：DELETE、INSERT、UPDATE。

触发器必须指定在语句执行之前还是之后自动执行，之前执行使用 BEFORE 关键字，之后执行使用 AFTER 关键字。BEFORE 用于数据验证和净化，AFTER 用于审计跟踪，将修改记录到另外一张表中。

INSERT 触发器包含一个名为 NEW 的虚拟表。

```
CREATE TRIGGER mytrigger AFTER INSERT ON mytable
FOR EACH ROW SELECT NEW.col into @result;

SELECT @result; -- 获取结果
```

DELETE 触发器包含一个名为 OLD 的虚拟表，并且是只读的。

UPDATE 触发器包含一个名为 NEW 和一个名为 OLD 的虚拟表，其中 NEW 是可以被修改的，而 OLD 是只读的。

MySQL 不允许在触发器中使用 CALL 语句，也就是不能调用存储过程

事务管理

基本术语：

- 事务(transaction)指一组 SQL 语句；
- 回退(rollback)指撤销指定 SQL 语句的过程；
- 提交(commit)指将未存储的 SQL 语句结果写入数据库表；
- 保留点(savepoint)指事务处理中设置的临时占位符(placeholder)，你可以对它发布回退(与回退整个事务处理不同)。

不能回退 SELECT 语句，回退 SELECT 语句也没意义；也不能回退 CREATE 和 DROP 语句。

MySQL 的事务提交默认是隐式提交，每执行一条语句就把这条语句当成一个事务然后进行提交。当出现 START TRANSACTION 语句时，会关闭隐式提交；当 COMMIT 或 ROLLBACK 语句执行后，事务会自动关闭，重新恢复隐式提交。

通过设置 autocommit 为 0 可以取消自动提交；autocommit 标记是针对每个连接而不是针对服务器的。

如果没有设置保留点，ROLLBACK 会回退到 START TRANSACTION 语句处；如果设置了保留点，并且在 ROLLBACK 中指定该保留点，则会回退到该保留点。

```
START TRANSACTION
// ...
SAVEPOINT delete1
// ...
ROLLBACK TO delete1
// ...
COMMIT
```

字符集

基本术语：

- 字符集为字母和符号的集合；
- 编码为某个字符集成员的内部表示；
- 校对字符指定如何比较，主要用于排序和分组。

除了给表指定字符集和校对外，也可以给列指定：

```
CREATE TABLE mytable
(col VARCHAR(10) CHARACTER SET latin COLLATE latin1_general_ci )
DEFAULT CHARACTER SET hebrew COLLATE hebrew_general_ci;
```

可以在排序、分组时指定校对：

```
SELECT *
FROM mytable
ORDER BY col COLLATE latin1_general_ci;
```

权限管理

MySQL 的账户信息保存在 mysql 这个数据库中。

```
USE mysql;
SELECT user FROM user;
```

创建账户

新创建的账户没有任何权限。

```
CREATE USER myuser IDENTIFIED BY 'mypassword';
```

修改账户名

```
RENAME myuser TO newuser;
```

删除账户

```
DROP USER myuser;
```

查看权限

```
SHOW GRANTS FOR myuser;
```

授予权限

账户用 username@host 的形式定义，username@% 使用的是默认主机名。

```
GRANT SELECT, INSERT ON mydatabase.* TO myuser;
```

删除权限

GRANT 和 REVOKE 可在几个层次上控制访问权限：

- 整个服务器，使用 GRANT ALL 和 REVOKE ALL；
- 整个数据库，使用 ON database.*；
- 特定的表，使用 ON database.table；

- 特定的列;
- 特定的存储过程。

```
REVOKE SELECT, INSERT ON mydatabase.* FROM myuser;
```

更改密码

必须使用 Password() 函数

```
SET PASSWROD FOR myuser = Password('new_password');
```

🤔👀SQL语言 - SQL语句练习

构建如下表结构

还有一个Grade表，在如下的练习中体现

插入数据

下面表SQL和相关测试数据是我Dump出来的

```
-- MySQL dump 10.13  Distrib 5.7.17, for macos10.12 (x86_64)
--
-- Host: localhost    Database: learn_sql_pdai_tech
-- ------------------------------------------------------
-- Server version   5.7.28

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table `COURSE`
--

DROP TABLE IF EXISTS `COURSE`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `COURSE` (
  `CNO` varchar(5) NOT NULL,
  `CNAME` varchar(10) NOT NULL,
  `TNO` varchar(10) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `COURSE`
--

LOCK TABLES `COURSE` WRITE;
/*!40000 ALTER TABLE `COURSE` DISABLE KEYS */;
INSERT INTO `COURSE` VALUES ('3-105','计算机导论','825'),('3-245','操作系统','804'),('6-
166','数据电路','856'),('9-888','高等数学','100');
/*!40000 ALTER TABLE `COURSE` ENABLE KEYS */;
UNLOCK TABLES;

--
```

```
-- Table structure for table `SCORE`
--

DROP TABLE IF EXISTS `SCORE`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `SCORE` (
  `SNO` varchar(3) NOT NULL,
  `CNO` varchar(5) NOT NULL,
  `DEGREE` decimal(10,1) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `SCORE`
--

LOCK TABLES `SCORE` WRITE;
/*!40000 ALTER TABLE `SCORE` DISABLE KEYS */;
INSERT INTO `SCORE` VALUES ('103','3-245',86.0),('105','3-245',75.0),('109','3-
245',68.0),('103','3-105',92.0),('105','3-105',88.0),('109','3-105',76.0),('101','3-
105',64.0),('107','3-105',91.0),('101','6-166',85.0),('107','6-106',79.0),('108','3-
105',78.0),('108','6-166',81.0);
/*!40000 ALTER TABLE `SCORE` ENABLE KEYS */;
UNLOCK TABLES;

--
-- Table structure for table `STUDENT`
--

DROP TABLE IF EXISTS `STUDENT`;
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `STUDENT` (
  `SNO` varchar(3) NOT NULL,
  `SNAME` varchar(4) NOT NULL,
  `SSEX` varchar(2) NOT NULL,
  `SBIRTHDAY` datetime DEFAULT NULL,
  `CLASS` varchar(5) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `STUDENT`
--

LOCK TABLES `STUDENT` WRITE;
/*!40000 ALTER TABLE `STUDENT` DISABLE KEYS */;
INSERT INTO `STUDENT` VALUES ('108','曾华','男','1977-09-01 00:00:00','95033'),
('105','匡明','男','1975-10-02 00:00:00','95031'),('107','王丽','女','1976-01-23
00:00:00','95033'),('101','李军','男','1976-02-20 00:00:00','95033'),('109','王
芳','女','1975-02-10 00:00:00','95031'),('103','陆君','男','1974-06-03
00:00:00','95031');
/*!40000 ALTER TABLE `STUDENT` ENABLE KEYS */;
UNLOCK TABLES;

--
-- Table structure for table `TEACHER`
--

DROP TABLE IF EXISTS `TEACHER`;
```

```
/*!40101 SET @saved_cs_client     = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `TEACHER` (
  `TNO` varchar(3) NOT NULL,
  `TNAME` varchar(4) NOT NULL,
  `TSEX` varchar(2) NOT NULL,
  `TBIRTHDAY` datetime NOT NULL,
  `PROF` varchar(6) DEFAULT NULL,
  `DEPART` varchar(10) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
/*!40101 SET character_set_client = @saved_cs_client */;

--
-- Dumping data for table `TEACHER`
--

LOCK TABLES `TEACHER` WRITE;
/*!40000 ALTER TABLE `TEACHER` DISABLE KEYS */;
INSERT INTO `TEACHER` VALUES ('804','李诚','男','1958-12-02 00:00:00','副教授','计算机系'),('856','张旭','男','1969-03-12 00:00:00','讲师','电子工程系'),('825','王萍','女','1972-05-05 00:00:00','助教','计算机系'),('831','刘冰','女','1977-08-14 00:00:00','助教','电子工程系');
/*!40000 ALTER TABLE `TEACHER` ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

-- Dump completed on 2020-02-06 18:18:25
```

相关练习

- 1、 查询Student表中的所有记录的Sname、Ssex和Class列。

```
select SNAME, SSEX, CLASS from STUDENT;
```

- 2、 查询教师所有的单位即不重复的Depart列。

```
select distinct DEPART from TEACHER;
```

- 3、 查询Student表的所有记录。

```
select * from STUDENT;
```

- 4、 查询Score表中成绩在60到80之间的所有记录。

```
select *
from SCORE
where DEGREE > 60 and DEGREE < 80;
```

- 5、 查询Score表中成绩为85，86或88的记录。

```
select *
from SCORE
where DEGREE = 85 or DEGREE = 86 or DEGREE = 88;
```

- 6、 查询Student表中“95031”班或性别为“女”的同学记录。

```
select *
from STUDENT
where CLASS = '95031' or SSEX = '女';
```

- 7、 以Class降序查询Student表的所有记录。

```
select *
from STUDENT
order by CLASS desc;
```

- 8、 以Cno升序、Degree降序查询Score表的所有记录。

```
select *
from SCORE
order by CNO asc, DEGREE desc;
```

- 9、 查询“95031”班的学生人数。

```
select count(*)
from STUDENT
where CLASS = '95031';
```

- 10、查询Score表中的最高分的学生学号和课程号。

```
select
  sno,
  CNO
from SCORE
where DEGREE = (
  select max(DEGREE)
  from SCORE
);
```

- 11、查询‘3-105’号课程的平均分。

```
select avg(DEGREE)
from SCORE
where CNO = '3-105';
```

- 12、查询Score表中至少有5名学生选修的并以3开头的课程的平均分数。

```
select
  avg(DEGREE),
  CNO
from SCORE
where cno like '3%'
group by CNO
having count(*) > 5;
```

- 13、查询最低分大于70，最高分小于90的Sno列。

```
select SNO
from SCORE
group by SNO
having min(DEGREE) > 70 and max(DEGREE) < 90;
```

- 14、查询所有学生的Sname、Cno和Degree列。

```
select
  SNAME,
  CNO,
  DEGREE
from STUDENT, SCORE
where STUDENT.SNO = SCORE.SNO;
```

- 15、查询所有学生的Sno、Cname和Degree列。

```
select
  SCORE.SNO,
  CNO,
  DEGREE
from STUDENT, SCORE
where STUDENT.SNO = SCORE.SNO;
```

- 16、查询所有学生的Sname、Cname和Degree列。

```
SELECT
  A.SNAME,
  B.CNAME,
  C.DEGREE
FROM STUDENT A
  JOIN (COURSE B, SCORE C)
    ON A.SNO = C.SNO AND B.CNO = C.CNO;
```

- 17、查询“95033”班所选课程的平均分。

```
select avg(DEGREE)
from SCORE
where sno in (select SNO
              from STUDENT
              where CLASS = '95033');
```

- 18、假设使用如下命令建立了一个grade表：

```
create table grade (
  low  numeric(3, 0),
  upp  numeric(3),
  rank char(1)
);
insert into grade values (90, 100, 'A');
insert into grade values (80, 89, 'B');
insert into grade values (70, 79, 'C');
insert into grade values (60, 69, 'D');
insert into grade values (0, 59, 'E');
```

- 现查询所有同学的Sno、Cno和rank列。

```
SELECT
  A.SNO,
  A.CNO,
  B.RANK
FROM SCORE A, grade B
WHERE A.DEGREE BETWEEN B.LOW AND B.UPP
ORDER BY RANK;
```

- 19、查询选修“3-105”课程的成绩高于“109”号同学成绩的所有同学的记录。

```
select *
from SCORE
where CNO = '3-105' and DEGREE > ALL (
  select DEGREE
  from SCORE
  where SNO = '109'
);
```

- 20、查询score中选学一门以上课程的同学中分数为非最高分成绩的学生记录

```
select * from STUDENT where SNO
  in (select SNO
  from SCORE
  where DEGREE < (select MAX(DEGREE) from SCORE)
  group by SNO
  having count(*) > 1);
```

- 21、查询成绩高于学号为“109”、课程号为“3-105”的成绩的所有记录。

```
select *
from SCORE
where CNO = '3-105' and DEGREE > ALL (
  select DEGREE
  from SCORE
  where SNO = '109'
);
```

- 22、查询和学号为108的同学同年出生的所有学生的Sno、Sname和Sbirthday列。

```
select
  SNO,
  SNAME,
  SBIRTHDAY
from STUDENT
where year(SBIRTHDAY) = (
  select year(SBIRTHDAY)
  from STUDENT
  where SNO = '108'
);
```

- 23、查询“张旭“教师任课的学生成绩。

```
select *
from SCORE
where cno = (
  select CNO
  from COURSE
    inner join TEACHER on COURSE.TNO = TEACHER.TNO and TNAME = '张旭'
);
```

- 24、查询选修某课程的同学人数多于5人的教师姓名。

```
select TNAME
from TEACHER
where TNO = (
  select TNO
  from COURSE
  where CNO = (select CNO
               from SCORE
               group by CNO
               having count(SNO) > 5)
);
```

- 25、查询95033班和95031班全体学生的记录。

```
select *
from STUDENT
where CLASS in ('95033', '95031');
```

- 26、查询存在有85分以上成绩的课程Cno.

```
select cno
from SCORE
group by CNO
having MAX(DEGREE) > 85;
```

- 27、查询出“计算机系“教师所教课程的成绩表。

```
select *
from SCORE
where CNO in (select CNO
              from TEACHER, COURSE
              where DEPART = '计算机系' and COURSE.TNO = TEACHER.TNO);
```

- 28、查询“计算机系”与“电子工程系“不同职称的教师的Tname和Prof

```
select
  tname,
  prof
from TEACHER
where depart = '计算机系' and prof not in (
  select prof
  from TEACHER
  where depart = '电子工程系'
);
```

- 29、查询选修编号为“3-105“课程且成绩至少高于选修编号为“3-245”的同学的Cno、Sno和Degree,并按Degree从高到低次序排序。

```
select
  CNO,
  SNO,
  DEGREE
from SCORE
where CNO = '3-105' and DEGREE > any (
  select DEGREE
  from SCORE
  where CNO = '3-245'
)
order by DEGREE desc;
```

- 30、查询选修编号为“3-105”且成绩高于选修编号为“3-245”课程的同学的Cno、Sno和Degree.

```
SELECT *
FROM SCORE
WHERE DEGREE > ALL (
  SELECT DEGREE
  FROM SCORE
  WHERE CNO = '3-245'
)
ORDER by DEGREE desc;
```

- 31、查询所有教师和同学的name、sex和birthday.

```
select
  TNAME     name,
  TSEX      sex,
  TBIRTHDAY birthday
from TEACHER
union
select
  sname     name,
  SSEX      sex,
  SBIRTHDAY birthday
from STUDENT;
```

- 32、查询所有“女”教师和“女”同学的name、sex和birthday.

```
select
  TNAME     name,
  TSEX      sex,
  TBIRTHDAY birthday
from TEACHER
where TSEX = '女'
union
select
  sname     name,
  SSEX      sex,
  SBIRTHDAY birthday
from STUDENT
where SSEX = '女';
```

- 33、查询成绩比该课程平均成绩低的同学的成绩表。

```
SELECT A.*
FROM SCORE A
WHERE DEGREE < (SELECT AVG(DEGREE)
                FROM SCORE B
                WHERE A.CNO = B.CNO);
```

- 34、查询所有任课教师的Tname和Depart.

```
select
  TNAME,
  DEPART
from TEACHER a
where exists(select *
             from COURSE b
             where a.TNO = b.TNO);
```

- 35、查询所有未讲课的教师的Tname和Depart.

```
select
  TNAME,
  DEPART
from TEACHER a
where tno not in (select tno
                  from COURSE);
```

- 36、查询至少有2名男生的班号。

```
select CLASS
from STUDENT
where SSEX = '男'
group by CLASS
having count(SSEX) > 1;
```

- 38、查询Student表中每个学生的姓名和年龄。

```
select
  SNAME,
  year(now()) - year(SBIRTHDAY)
from STUDENT;
```

- 39、查询Student表中最大和最小的Sbirthday日期值。

```
select min(SBIRTHDAY) birthday
from STUDENT
union
select max(SBIRTHDAY) birthday
from STUDENT;
```

- 40、以班号和年龄从大到小的顺序查询Student表中的全部记录。

```
select *
from STUDENT
order by CLASS desc, year(now()) - year(SBIRTHDAY) desc;
```

- 41、查询“男”教师及其所上的课程。

```
select *
from TEACHER, COURSE
where TSEX = '男' and COURSE.TNO = TEACHER.TNO;
```

- 42、查询最高分同学的Sno、Cno和Degree列。

```
select
  sno,
  CNO,
  DEGREE
from SCORE
where DEGREE = (select max(DEGREE)
                from SCORE);
```

- 43、查询和“李军”同性别的所有同学的Sname.

```
select sname
from STUDENT
where SSEX = (select SSEX
              from STUDENT
              where SNAME = '李军');
```

- 44、查询和“李军”同性别并同班的同学Sname.

```
select sname
from STUDENT
where (SSEX, CLASS) = (select
                         SSEX,
                         CLASS
                       from STUDENT
                       where SNAME = '李军');
```

- 45、查询所有选修“计算机导论”课程的“男”同学的成绩表

```
select *
from SCORE, STUDENT
where SCORE.SNO = STUDENT.SNO and SSEX = '男' and CNO = (
  select CNO
  from COURSE
  where CNAME = '计算机导论');
```

- 46、使用游标方式来同时查询每位同学的名字，他所选课程及成绩。

```
declare
 cursor student_cursor is
  select S.SNO,S.SNAME,C.CNAME,SC.DEGREE as DEGREE
  from STUDENT S, COURSE C, SCORE SC
  where S.SNO=SC.SNO
  and SC.CNO=C.CNO;

  student_row student_cursor%ROWTYPE;

begin
  open student_cursor;
   loop
    fetch student_cursor INTO student_row;
    exit when student_cursor%NOTFOUND;
     dbms_output.put_line( student_row.SNO || '' ||

student_row.SNAME|| '' || student_row.CNAME || '' ||

student_row.DEGREE);
   end loop;
  close student_cursor;
END;
/
```

- 47、 声明触发器指令，每当有同学转换班级时执行触发器显示当前和之前所在班级。

```
CREATE OR REPLACE TRIGGER display_class_changes
AFTER DELETE OR INSERT OR UPDATE ON student
FOR EACH ROW
WHEN (NEW.sno > 0)

BEGIN

   dbms_output.put_line('Old class: ' || :OLD.class);
   dbms_output.put_line('New class: ' || :NEW.class);
END;
/


Update student
```

```
set class=95031
where sno=109;
```

- 48、 删除已设置的触发器指令

```
DROP TRIGGER display_class_changes;
```