学习总结

- mysql中只有删除某些行的数据采用DELETE，其他删除表，删除约束都是用DROP
- 只有创建触发器时用了WEHN，其他时候基本都是WHERE

SQL数据定义

这里给它命名为数据定义是因为这里还没有到对某些行进行操作，还是对表的总体框架进行定义

schema和domain和table区别

1. **Schema**:
   - **定义**：Schema 是数据库中的一个逻辑容器，用于组织和管理数据库对象，例如表、视图、索引等。
   - **作用**：它提供了一种将相关对象组织在一起的方式，有助于管理和控制数据库对象的命名空间，以及对这些对象的访问权限和安全性进行管理。
   - **示例**：在一个数据库中，可以有多个 schema，每个 schema 包含一组相关的数据库对象。例如，可以有一个用于存储用户信息的 schema，另一个用于存储产品信息的 schema。
2. **Domain**:
   - **定义**：Domain 是数据库中用于约束数据类型和数据值的规范或范围。
   - **作用**：它定义了一个数据列可以接受的数据类型和值的范围，以确保数据的一致性和完整性。
   - **示例**：例如，可以定义一个 domain，规定一个列只能接受特定的字符串格式，或者规定一个列的值必须在某个范围内。
3. **Table**:
   - **定义**：Table 是数据库中的一个物理或逻辑结构，用于存储数据。
   - **作用**：它是数据库中最基本的数据组织单元，用于存储和组织相关数据。
   - **示例**：一个 table 可以包含多个列，每列代表一种数据类型，每行代表一个数据记录。例如，一个存储用户信息的 table 可能包含列如用户名、密码、电子邮件地址等。

创建表

```
CREATE TABLE tableName (
 {columnName dataType [NOT NULL] [UNIQUE]
  [DEFAULT defaultOption]
  [CHECK (searchCondition)]}[, ...]
 [PRIMARY KEY (listOfColumns),]
 {[UNIQUE (listOfColumns)][, ...]}
 {[FOREIGN KEY (listOfFKColumns)
   REFERENCES parentTableName [(listOfCKColumns)]
    [ON UPDATE referentialAction]
    [ON DELETE referentialAction]][, ...]}
 {[CHECK (searchConditions)][, ...]}
);
```

- 除了上述还可以添加CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Sandnes') 类似的CONSTRAINT

- REFERENCES意思就是参考的意思，也即是根据哪个主表中的列进行关联

修改表

语法

```
ALTER TABLE tableName
 [ADD newColumnName dataType]
 [ADD constraint]
 [DROP COLUMN columnName]
 [DROP constraint]
 [MODIFY columnName dataType];
```

- 新增加的列为空值
- 修改列定义可能破坏已有数据

创建索引

创建索引

语法

```
CREATE [UNIQUE] [CLUSTER] INDEX indexName
ON tabName(colName [ASC|DESC][, colName[ASC|DESC]] ...);
```

- 一条语句建立一个索引
- 排列次序缺省为ASC
- UNIQUE, 每个索引值对应唯一一条数据记录
- CLUSTER, 聚簇索引, 即索引项的顺序与表中记录的物理顺序一致
- 索引有维护代价

视图

视图

- 视图是虚关系：无需另外存在于数据库中, 根据某个特定需求而生成的虚关系
- 视图的内容基于一个或多个基本关系的查询
- 视图是动态结果：对一个或多个基本关系进行关系操作得到的动态结果

创建视图

视图可以从一张表、几张表或其他视图中创建

语法

```
CREATE VIEW viewName
[(columnName1 [, columnName2] ...)]
AS SELECT sentence
[WITH CHECK OPTION];
```

- 若视图名后的列名表与SELECT语句里的列名表相同，则视图名后的列名表可以省略
- 若使用WITH CHECK OPTION，则对视图进行INSERT和UPDATE操作时，保证行满足视图定义中的WHERE子句指定的条件

72

- 若视图成功更新，实际上这是对基本表的更新

SQL定义完整性

约束

- 断言约束监测时机可以是事务结束时或者每一条SQL语句执行完成时

一般约束 断言

断言约束可以涉及多个表

创建断言 语法

```
CREATE ASSERTION assertionName
CHECK (searchCondition);
```

触发器

触发器

触发器Trigger是一类靠事件驱动的特殊过程, 一旦由用户创建, 任何触发该触发器的事件发生且满足条件时, 系统自动激活相应的触发器, 执行相应动作

触发器定义

- 触发事件
- 触发时机
- 触发条件
- 触发动作

- 触发器用的是WHEN，其后面跟着是触发后的相应动作
- 这个OF是在指定了特定列才用，否则就直接是ON tabname
- BEGIN,END是确定一个语句块的边界，因为触发的相应动作可能不只一条语句

创建触发器 语法

```
CREATE TRIGGER trigger_name
{BEFORE|AFTER}
{INSERT|DELETE|UPDATE [OF colname[, colname ...]]}
ON tabname
[REFERENCING corr_name_def[, corr_name_def ...]]
[FOR EACH ROW|FOR EACH STATEMENT]
 [WHEN (search_condition)]
  {statement
  |BEGIN ATOMIC
    statement; [statement; ...]
   END};
```

- REFERENCING后跟这些

corr_name_def定义

```
{OLD [ROW] [AS] old_row_corr_name
|NEW [ROW] [AS] new_row_corr_name
|OLD TABLE [AS] old_table_corr_name
|NEW TABLE [AS] new_table_corr_name}
```

SQL数据操作

INSERT语句

- INSERT INTO ... VALUES
- insert语句显然要注意列与列要相互对应

INSERT语句

语法

```
INSERT INTO tblName [(colName1[, colName2] ...)]
VALUES (value1[, value2] ...);
```

```
INSERT INTO tbl1Name [(tbl1ColName1[, tbl1ColName2] ...)
 SELECT *|tbl2ColName1[, tbl2ColName2] ...
 FROM tbl2Name
 WHERE condition;
```

UPDATE语句

- UPDATE ... SET ...

UPDATE语句

一条UPDATE语句在某一时刻只能更新一张表, 但可以更新一张表中的多列, 也可以更新多行数据

语法

```
UPDATE tableName
SET column1 = value1[, column2 = value2] ...
[WHERE condition];
```

- WHERE子句决定需要更新的行
- 列名决定需要更新的列

DELETE语句

DELETE语句

一条DELETE语句可以从表中删除一行或多行整行数据, 不能从特定的列中删除数据

语法

```
DELETE FROM tableName
[WHERE condition];
```

简单查询

SELECT语句基本语法

```
SELECT [ALL|DISTINCT]
 {*|[colExpr [AS newName]][, ...]}
FROM tblName [alias][, ...]
[WHERE condition]
[GROUP BY colList [HAVING condition]]
[ORDER BY colList [ASC|DESC]];
```

SELECT语句含义

根据WHERE子句中的条件表达式，从FROM子句中的基本表或视图中找出满足条件的元组，可完成关系代数中的选择、投影、连接等运算

- SELECT子句，必须，列出目标字段，选出元组中的分量
- FROM子句，必须，指明所访问的表或视图
- WHERE子句，可选，查询条件，可以有多个
- GROUP BY子句，可选，将结果按字段分组，每个组产生结果表中的一个元组
- HAVING子句，可选，每组应满足的条件
- ORDER BY子句，可选，按指定的字段的值，以升序或降序排列查询结果

- ALL, 缺省值 DISTINCT, 消除结果中的重复行
- AS, 为列重命名
- BETWEEN AND / NOT BETWEEN AND ; BETWEEN并不能增强SQL的功能,等同于>= AND <=
- IN/NOT IN

IN / NOT IN

测试数据是否在/不在集合中

IN并不能增强SQL的功能, 但条件表达更简洁

例 列出所有的经理与主管

```
SELECT staffNo, fName, lName, position
FROM Staff
WHERE position IN ('Manager', 'Supervisor');

SELECT staffNo, fName, lName, position
FROM Staff
WHERE position = 'Manager' OR
      position = 'Supervisor';
```

- LIKE/NOT LIKE

LIKE / NOT LIKE

SQL中有两种特殊的模式匹配符

- %: 百分号表示零或多个字符序列
- _: 下划线表示任意单个字符

例 找出地址中含有字符串'Glasgow'的所有房主

```
SELECT ownerNo, fName, lName, address, telNo
FROM PrivateOwner
WHERE address LIKE '%Glasgow%';
```

- ESCAPE

ESCAPE, 定义转义符

若查找的字符串本身包含特殊模式匹配符, 用转义符

例 查询学号的第二个字符不是'%'的学生信息

```
SELECT *
FROM Student
WHERE sno NOT LIKE '_\%%' ESCAPE '\';
```

- IS NULL / IS NOT NULL

IS NULL / IS NOT NULL

- NULL表示值未知, 不是常量, 不能作为操作数,
  IS不能用等号=代替,
  IS NOT不能用不等号!=或<>代替
- NULL不能用来表示缺省值
- 大部分聚集函数忽略空值
- 尽量少用空值

例 列出查看过编号为PG4的房产但没有留下意见的客户信息

```
SELECT clientNo, viewDate
FROM Viewing
WHERE propertyNo = 'PG4' AND comment IS NULL;
```

- 排序

ORDER BY子句 语法

```
ORDER BY columnList [ASC|DESC][, ...]
```

- ASC, 升序(缺省); DESC, 降序
- 第一项相同时，再按照第二项进行排序
- ORDER BY子句须是SELECT语句的最后一个子句

多列排序

例 产生按类型和租金排序的一个房产简要报表

```
SELECT propertyNo, type, rooms, rent
FROM PropertyForRent
ORDER BY type, rent DESC;
```

propertyNo	type	rooms	rent
PG16	Flat	4	450
PL94	Flat	4	400
PG36	Flat	3	375
PG4	Flat	3	350
PA14	House	6	650
PG21	House	5	600

聚合函数

- 要注意聚合函数的使用限制

聚合函数

ISO标准定义了五个聚合函数

名称	参数列数据类型	结果类型	描述
COUNT	任意, 可以是*	数值型	列中数据计数
SUM	数值型	数值型	列中数据总和
AVG	数值型	数值型	列中数据平均值
MAX	字符型或数值型	同参数列类型一样	列中数据最大值
MIN	字符型或数值型	同参数列类型一样	列中数据最小值

- 只能用于SELECT列表和HAVING子句中
- 只对表中单个列的数据进行操作, 返回一个值

GROUP BY

- 挺重要的一个概念，必考

分组

GROUP BY子句 语法

```
GROUP BY columnName[, ...] [HAVING condition]
```

- 将查询结果按指定列名的取值分组; 并选取满足HAVING中的聚合函数表达式所指定条件的组
- SELECT子句仅可包含列名、聚合函数、常量、组合前述各项的表达式
- SELECT子句中除集合函数外, 列须在GROUP BY子句中出现
- 应用GROUP BY时, 两个空值是相等的

- 注意理解这句话：SELECT子句中除集合函数外，列必须在GROUP BY子句中出现

> 例 找出工作在每一个分支机构的职员人数和他们的工资总和
>
> ```
> SELECT branchNo,
>        COUNT(staffNo) AS myCount,
>        SUM(salary) AS mySum
> FROM Staff
> GROUP BY branchNo;
> ```
>
> | branchNo | myCount | mySum |
> |---|---|---|
> | B003 | 3 | 54000.00 |
> | B005 | 2 | 39000.00 |
> | B007 | 1 | 9000.00 |

HAVING

- HAVING语句,**必须和GROUP BY语句合在一起使用**
- 注意理解第二个概念，其实GROUP BY就很像分组排序那种先分组再对每个组进行查询，**然后每组得到一个结果**

HAVING子句

- 包含在GROUP BY子句中, 不可单独使用, 约束哪些分组出现在最终查询结果中
- WHERE子句将单行过滤到查询结果中; 而HAVING子句则将分组过滤到查询结果中
- HAVING子句使用的列名必须已出现在GROUP BY子句列表中或者包括在聚合函数中
- 实践中, HAVING子句的条件表达式至少包括一个聚合函数, 否则可把条件移到WHERE子句中先过滤单行

- 例子

例 对于职员人数多于1人的分支机构, 计算出每一个分支机构的职员人数和他们的工资总和

```
SELECT branchNo,
       COUNT(staffNo) AS myCount,
       SUM(salary) AS mySum
FROM Staff
GROUP BY branchNo HAVING COUNT(staffNo) > 1
ORDER BY branchNo;
```

branchNo	myCount	mySum
B003	3	54000.00
B005	2	39000.00