一、基础

1 常用操作

注：在实际应用中，SQL关键字最好使用大写，表名使用小写，以增强SQL代码可读性。

1.1 连接管理

```
mysql -u root -p #连接本地MySQL服务器

```

1.2 常用show命令

```
HELP SHOW; #查看show命令的帮助文档
SHOW CREATE DATABASE <datanaseName>; #查看创建数据库的sql语句
SHOW CREATE TABLE <tableName>; #查看创建表的sql语句
```

1.3 数据库服务器管理

```
SHOW STATUS; #查看广泛的服务器信息
SHOW ERROS/WARNINGS; #查看服务器错误或警告信息
```

1.4 用户管理

```
UPDATE mysql.user SET password=password("hbd1993720")
WHERE user='root'; #修改用户密码
flush privileges;

SHOW GRANTS; #查看授予用户的安全权限
```

1.5 数据库操作

```
SHOW DATABASES; #查看所有数据库
use <datanaseName>; #进入数据库
SHOW CREATE DATABASE <datanaseName>; #查看创建数据库的sql语句
CREATE DATABASE <databaseName>; #创建数据库
```

1.6 基础表操作

```
SHOW TABLES;      #查看数据库中的表
SHOW CREATE TABLE <tableName>; #查看创建表的sql语句
DESC <tableName>;    #查看表的列信息
 == DESCRIBLE <tableName>;
 == SHOW COLUMNS from <tableName>;
CREATE TABLE student(id int, name text);    #创建新表
INSERT INTO student values(1,"hunk");       #插入一条记录
```

2 SQL查询基础

2.1 基本排序过滤

```
###############
检索数据
###############
SELECT * FROM student;    #查询表中所有记录，列的顺序一般是列在表定义中出现的顺序
//注：检索不需要的列通常会降低检索和应用程序的性能，慎用通配符检索
SELECT id,name FROM student;    #查询表中所有记录的部分列
SELECT DISTINCT name FROM student; #对查询结果去重,DISTINCT应用于所有列，除非指定的所有列
#都不同，否则所有行都将被检索出来
//限制检索结果行数
SELECT id,name FROM student LIMIT 5; #限制返回结果不多于5行，从第一行开始，也就是行0
SELECT id,name FROM student LIMIT 5,6; #限制返回结果不多于6行，从行5开始

###############
检索结果排序
###############
//ORDER BY子句应该位于FROM之后
SELECT name FROM student ORDER BY name; #对检索结果进行排序，允许用非检索的列来排序
SELECT name FROM student
ORDER BY name DESC; #按降序排序，默认是升序
//DESC关键字只应用到直接位于其前面的列名
SELECT name,dept_name FROM student
ORDER BY name,dept_name; #只有在name相同时，才根据dept_name排序
SELECT name,dept_name FROM student
ORDER BY name DESC,dept_name; #按name降序排序，name相同时按dept_name升序排序
```

```
//使用ORDER BY和LIMIT可以找出某个最高或最低的值，LIMIT必须位于ORDER BY之后
SELECT price FROM product
ORDER BY price DESC LIMIT 1; #检索价格最高的产品

###############
过滤检索结果
###############
//同时使用WHERE子句和ORDER BY子句时，ORDER BY子句应该位于WHERE之后，否则将会产生错误
/1.匹配检查/
SELECT name,dept_name FROM student
WHERE name = 'Brown'; #检索name等于..的行

/2.不匹配检查/
SELECT name,dept_name FROM student
WHERE name <> 'Brown'; #检索name不等于..的行，<>等于!=

/3.范围检索/
SELECT name,dept_name FROM student
WHERE name BETWEEN 'Brandt' AND 'Sanchez'
ORDER BY name;

/4.空值检查/
SELECT name,dept_name FROM student
WHERE name IS NULL;
//注意：通过过滤条件选择出不具有特定值的行时，我们可能希望返回具有NULL值得行。但是因为具有特殊的含义，
//数据库不知道它们是否匹配，所以在匹配过滤或不匹配过滤时不返回它们，所以可能需要额外检查。

/5.组合过滤/
//注意：AND操作符优先级高于OR，组合使用时需要避免错误组合，必要时使用()来指定组合顺序
SELECT name,dept_name FROM student
WHERE name = 'Levy'
AND dept_name = 'Physics'; #AND操作符
SELECT name,dept_name FROM student
WHERE name = 'Levy'
OR dept_name = 'Physics'; #OR操作符

/6.IN操作符/
//IN操作符后跟一个清单，作用类似于OR操作符，相比OR其有如下特点：
```

```
//a.有长的合法清单时，IN的语法更清晰直观
//b.计算次序更容易管理，因为使用的操作符少
//c.IN操作符一般比OR操作符清单执行更快
//d.IN的最大优点是可以包含其他SELECT语句，使得能够动态创建WHERE语句
SELECT name,dept_name FROM student
WHERE name IN ('Brandt', 'Levy'); #检索name等于Brandt或者Levy

/7.NOT操作符/
//NOT操作符否定跟在它之后的条件
SELECT name,dept_name FROM student
WHERE name IN ('Brandt', 'Levy'); #检索name不等于Brandt和Levy
```

2.2 通配符和正则表达式过滤

```
/8.通配符过滤/
//通配符就是一些特殊字符，搜索模式就是字面值、通配符或者他们的组合，比如'B%n%',
//为在搜搜子句中使用通配符，必须使用LIKE操作符后面
//8.1 百分号(%)通配符
// %可以匹配0个、1个和多个任意字符，但是不能匹配到NULL
SELECT * FROM student
WHERE name LIKE 'B%n%';
//8.2 下划线(_)通配符
// _只匹配单个字符，不能多也不能少
//注意：通配符搜索的处理一般要比前面的其他搜索花更多的时间，一些使用建议：
//1.不要过度使用通配符。如果其他操作能达到目的，尽量不用通配符。
//2.必须使用通配符时，最后不要把通配符放在搜索模式开头，比如'%hunk'，这样搜索最慢。***
//3.注意统配符位置，否则搜索结果达不到预期。

/9.正则表达式搜索/
//正则表达式是用来匹配文本的字符集合，MySQL支持的正则表达式只是正则表达式语言的子集
//9.1 与通配符的区别-- LIKE匹配整个串，而REGEXP匹配子串
SELECT * FROM student
WHERE dept_name LIKE 'sic'; #只匹配值等于sic的行

SELECT * FROM student
WHERE dept_name REGEXP 'sic'; #匹配值包含sic的行

SELECT * FROM student
WHERE dept_name REGEXP '.sic'; #'.'需要至少匹配到一个字符，列值以sic开头的行无法匹配到
```

```
26
27 //9.2 MySQL中的正则表达式不区分大小写，可以加上BINARY关键字来区分
28 SELECT * FROM student
29 WHERE dept_name REGEXP BINARY '.SIC';
30
31 //9.3 OR匹配
32 SELECT * FROM student
33 WHERE dept_name REGEXP 'sic | anc'; #匹配值包含sic或者anc的行
34
35 //9.4 用[]匹配任意单一字符
36 //[]里面的多个字符是或的关系，[ab] == [a|b]
37 SELECT * FROM student
38 WHERE dept_name REGEXP '^[MP].sic'; #匹配列值以M或者P开头的行
39
40 //除非把字符|括在一个集合中，否则它将应用于整个串
41 //字符集合也可以被否定，在集合开始处加上^即可
42 SELECT * FROM student
43 WHERE dept_name REGEXP '^[^MP].sic'; #匹配列值不以M和P开头的行
44
45 //9.5 匹配范围
46 [0123456789] == [0-9]
47 [a-z] #匹配任意字母字符
48
49 // 9.6 匹配特殊字符
50 //前面使用的. | [] - 都是正则表达式的特殊字符，如果需要把这些特殊字符作为普通字符用于匹配
51 //需要在前面加上\\来做转义， 比如\\.表示查找.，匹配反斜杠本身就需要用\\\
52 //()也需要用\\(\\)来转义
53 //多数正则表达式实现使用\来转义，MySQL要求两个反斜杠，自己处理一个，正则表达式库解释一个
54
55 //9.7 预定义字符集
56 [:alnum:] #任意字母和数字，等于[a-zA-Z0-9]
57 ...
58
59 //9.8 匹配多个实例
60 * #0个或者多个匹配
61 + #1个或者多个匹配，等于{1,}
62 ? #0个或1个匹配，等于{0,1}
63 {n} #指定数目的匹配
64 {n,} #不少于指定数目的匹配
```

```
{n,m} #匹配数目的范围,m不超过255

//9.9 定位符
^ #文本开始
$ #文本结尾
[[:<:]] #词的开始
[[:<:]] #词的结尾
```

2.3 计算字段

```
/10.创建计算字段/
//数据库表中的列基本上是固定的，但是客户程序往往并不直接使用查询表的结果，而是需要做额外的处理：
//1.对多个列的数据进行算数运算得到新的数据
//2.将做个字段拼接成新的字段，比如省、市、区、街道拼接成最终地址
//如果在SQL中完成这些运算，直接生成目标数据，相比客户程序来二次处理更为高效，毕竟DBMS做了很多优化工作
SELECT Concat( #Concat()函数用于拼接字符串
Rtrim(name), #Rtrim()函数去掉字符串右边的空格
'(',
Ltrim(dept_name),
 ')') AS name_dept, #AS将拼接后的结果命名为新的字段（也叫导出列），客户程序可以直接使用
 Now() as time #Now()函数获取当前系统时间
 FROM student
 WHERE dept_name
 REGEXP '^[^MP].';

//MySQL支持的算数操作符：+ - * /
```

2.4 数据处理和聚集函数

```
/11.数据处理函数/
//MySQL内置的数据处理函数（部分，包括前面使用的Rtrim()和Ltrim()）：
Upper() #文本转换成大写
Lower() #将串转换成小写
Length() #计算串的长度
Locate() #找出串的一个子串
Right() #返回串右边的字符，这个函数可能不太好使
Left() #返回串左边的字符余数
Abs() #返回一个数的绝对值
 Mod() #返回除操作的余数

```

```
//日期和时间函数
Now() #返回当前日期和时间，格式是MySQL存储时间的标准格式：2019-05-17 12:38:52
Date() #从标准格式中取出日期，Date("2019-05-17 12:38:52") -> 2019-05-17
Year() #从标准格式中取出年份
Month() #从标准格式中取出月份
Time() #从标准格式中取出时间

//比如检索2005年9月份的所有订单
SELECT id,order_num
FROM orders
WHERE Date(order_date)
BETWEEN '2005-09-01' AND '2005-09-30';
或者
SELECT id,order_num
FROM orders
WHERE Year(order_date) = 2005
Month(order_date) = 9; #这种更方便


```

```
/12.聚集函数/
AVG() #返回列的平均值,只能用于确定特定数值列的平均值，多个列使用多个AVG函数，忽略值为NULL的行
COUNT() #返回某列的行数，COUNT(*)：对表中行的数目计数，包括NULL值；
 #COUNT(column)：对特定列具有值得行计数，忽略NULL值
MAX() #返回某列的最大值，可以是数值、日期甚至文本列，忽略值为NULL的行
MIN() #返回某列的最小值,与MAX类似，忽略值为NULL的行
SUM() #返回某列值之和
```

2.5 分组数据

```
/13.分组数据/
//13.1 创建分组
//比如返回每个系的名字及该系学生数目
SELECT dept_name,
COUNT(*) AS count from student //也可以用COUNT(dept_name)
GROUP BY dept_name;

//1.GROUP BY子句必须出现在WHERE子句之后，ORDER BY子句之前
//2.除聚集计算语句外，SELECT语句中的每个列都必须在GROUP BY子句中给出
```

```
//13.2 过滤分组
//HAVING可以代替WHERE子句的功能,区别：WHERE过滤航，HAVING过滤分组,或者说WHERE在数据分组前过滤
//HAVING在数据分组后进行过滤
SELECT dept_name,
COUNT(*) AS count FROM student
GROUP BY dept_name
HAVING count >= 2; #找出学生超过两名的系
等同于
SELECT dept_name,
 COUNT(*) AS count FROM student
 GROUP BY dept_name
 HAVING COUNT(*) >= 2;

 //排序输出结果，一般在使用GROUP BY子句时，应该也给出ORDER BY子句，不要依赖于GROUP BY排序数据
 SELECT dept_name,
 COUNT(*) AS count FROM student
 GROUP BY dept_name
 HAVING COUNT(*) >= 2 ORDER by count;

 //HAVING和WHERE同时使用的场景
 SELECT vend_id, COUNT(*) AS num_prods
 FROM products WHERE prod_price >= 10
 GROUP BY vend_id
 HAVING COUNT(*) >= 2; #计算出具有2个（含）以上，价格为10（含）以上的产品的供应商
```

2.6 子查询

```
/14. 子查询/
//嵌套子查询，建议写成下面缩进格式以更好区分各级子查询
//对于嵌套的子查询数目没有限制，不过在实际使用中由于性能限制，不能嵌套太多的子查询
#查询所属Tylor这栋楼里的系的学生的选课情况，结果包含学生ID和所选课程ID
SELECT ID, course_id
FROM takes WHERE ID IN(SELECT ID
 FROM student
 WHERE dept_name IN (SELECT dept_name
 FROM department
  WHERE building = 'Taylor'));

```

```
//作为计算字段使用子查询
SELECT ID, name,
(
SELECT COUNT(*) FROM takes
WHERE student.ID = takes.ID #这里必须使用相关子查询（加上表名来区分ID）
)
AS take_num #子查询结果作为计字段
FROM student ORDER BY take_num; #查询每个学生选课的数目
```

2.7 联结表

```
/1. 内部联结 ****/
//dmeo:使用联结表代替前面的子查询写法：Tylor这栋楼里的系的学生的选课情况
SELECT takes.ID, course_id
FROM takes, student, department
WHERE takes.ID = student.ID
AND student.dept_name = department.dept_name
AND department.building = 'Taylor';
或者
SELECT takes.ID, course_id
 FROM takes
 INNER JOIN student
 INNER JOIN department
 ON takes.ID = student.ID #这里ON和WHERE是可以互换,但是没有INNER JOIN只能用WHERE
 AND student.dept_name = department.dept_name
 AND department.building = 'Taylor';
```

```
Notes:
1.上面两种写法都是属于等值联结（多个表联结时基于给定列做等值匹配），也称为内部联结
2.如果没有上面的联结条件，返回得则是笛卡尔积：第一张表的每一行与第二张表的每一个行做匹配，
3.笛卡尔积检索出的行数是多个表的行数的乘积，一般情况下这都不是我们需要的。
```

```
/2.外部联结/
//与内部联结对应，内部联结包括没有关联行的行
//demo:查询各个系选课的学生的人数
SELECT dept_name, COUNT(*) as num
FROM student
RIGHT OUTER JOIN takes #右外联结，takes表中的每一个行都会被关联
```

```
ON student.ID = takes.ID
GROUP BY dept_name;
等同于
 SELECT dept_name, COUNT(*) as num
 FROM student
 INNER JOIN takes #等值联结
 ON student.ID = takes.ID
 GROUP BY dept_name;
 不等于
 SELECT dept_name, COUNT(*) as num
 FROM student
 LEFT OUTER JOIN takes #左外联结，student表中的每一个行都会被关联，包括了没有选课的学生
 ON student.ID = takes.ID
 GROUP BY dept_name;
```

2.8 组合查询

```
//使用UNION组合两个SELECT查询，两个查询返回的结构是类似
SELECT ID, course_id, year
FROM takes WHERE course_id = 'CS-101'
UNION
SELECT ID, course_id, grade
FROM takes WHERE grade = 'A';
```

```
Notes:
1.UNION必须由两条以上的SELECT语句组成，语句之间用UNION分隔
2.UNION中的每个查询必须包含相同的列、表达式或聚集函数
3.列数据类型必须兼容：类型不必完全相同，但必须是DBMS可以隐含转换的类型
4.使用UINON时多个查询结果中重复的行被自动取消，可以使用UNION ALL避免自动取消
5.使用UNION时只能在最后的SELECT语句后面使用一次ORDER BY子句，实际上是整体排序。
```

2.9 全文本搜索

MyISAM引擎支持全文本搜索，InnoDB不支持。

为了进行全文本搜索，必须索引被搜索的列，而且随着数据的改变不断地重新索引。

在对表列进行适当设计后，MySQL会自动进行所有的索引和重新索引。

一般在创建表时启用全文本搜索：

```
CREATE TABLE test
 (ID int AUTO_INCREMENT,
```

```
3  name varchar(8) NOT NULL DEFAULT 'test',
4  address varchar(30) NOT NULL,
5  PRIMARY KEY(ID),
6  FULLTEXT(address)
7  )ENGINE=MyISAM;
```

MySQL根据FULLTEXT()的指示对列进行索引，也可以指定多个列。在定义后，MySQL自动维护该索引。
在增加、更新和删除行时，索引随之自动更新。

Notes:
也可以在创建表完成以后指定FULLTEXT，但是这种情况下所有已有数据必须立即索引。因此，
不要在导入数据时使用FULLTEXT，更新索引要花时间。如果正在导入数据到一个新表，此时不应该启用FULLTEXT索引，应该首先导入所有数据，然后再修改表，定义FULLTEXT。
这样有助于更快地导入数据。

索引之后，在WHERE子句中使用两个函数Match()和Against()执行全文本搜索。

```
1  SELECT address FROM test WHERE Match(address) Against('enyang');
```

全文本搜索的一个重要部分就是对结果排序，具有较高等级的行先返回，比如在第3个字符位置
匹配到的行优先级高于在第20个字符匹配到的行。
搜索不区分大小写，除非使用BINARY方式。
传递给Match()的值必须与FULLTEXT()定义的相同。如果指定多个列，则必须列出它们且次序正确

布尔全文本搜索
没有FULLTEXT索引也可以使用布尔搜索，但是这种操作比较缓慢。

```
1  SELECT address FROM test WHERE Match(address) Against('enyang' IN BOOLEAN
MODE);
```

布尔全文本搜索支持很多实用操作符：

```
1  + #包含，词必须存在
2  - #排除，词必须不出现
3  > #包含，而且增加等级值
4  < #包含，且减少等级值
5  () #把词组成表达式
```

```
~ #取消一个词的排序值
* #词尾的通配符
"" #定义一个短语
```

```
SELECT address FROM test WHERE Match(address) Against('+eny -ang*' IN BOOLEAN MODE);
#匹配包含词eny但不包含以ang开始的行
```

```
SELECT address FROM test WHERE Match(address) Against('+eny +ang' IN BOOLEAN MODE);
#匹配包含词eny和ang的行，必须匹配两个词
```

```
SELECT address FROM test WHERE Match(address) Against('eny ang' IN BOOLEAN MODE);
#匹配包含词eny或者ang的行，至少匹配1个词
```

```
SELECT address FROM test WHERE Match(address) Against('>eny <ang' IN BOOLEAN MODE);
#匹配包含词eny和ang的行，增加前者优先级，降低后者优先级
```

```
SELECT address FROM test WHERE Match(address) Against('"eny ang"' IN BOOLEAN MODE);
#匹配包含词"eny ang"的行
```

全文本搜索的使用说明

1. 索引全文本数据时，短词（3个或3个以下字符的词，可以设置长度约束）被忽略且从索引中排除
2. MySQL带有一个内建的非用词表（stopword），这些词在索引全文本数据时总是被忽略。
3. 许多词出现频率很高，搜索它们返回太多结果。MySQL规定一个词出现50%以上的行中，作为非用词忽略。
4. 这条50%规则不用于IN BOOLEAN MODE。
5. 如果表中的行少于3行，则全文本搜索不返回结果（因为每个词不出现或至少出现在50%的行中）。
6. 忽略词的单引号。don't索引为dont。

2.10 索引

```
SHOW INDEX FROM <table>; #查询表索引
```

3 数据插入和更新

3.1 INSERT语句

```
INSERT INTO
student(ID, name, dept_name, tot_cred) #列出所有或者部分必要的列更为安全
VALUES
('00001', 'Hunk', 'Comp. Sci.', 100),
('00002', 'Jack', 'Comp. Sci.', 200);
#如果ID可以由MySQL自己填充（自增长），可以不填充
#在一条INSERT语句插入多条记录相比于执行多条INSERT语句更为高效
```

```
INSERT INTO
student(ID, name, dept_name, tot_cred)
SELECT
ID, name, dept_name, tot_cred
FROM student_new;
#从其它表中查询出来直接插入表中，一般来讲两个表对应的列的属性应该一致，
#实际上，MySQL是利用位置来插入数据的，因此从其他不同列名的表中导入数据也是被允许的
```

Notes：执行客户端SQL语句的优先级由MySQL来决定，因为INSERT语句往往比较耗时，可能降低等待处理的SELECT语句的性能，如果数据查询更为重要（通常如此），可以在INSERT
和INTO之间加入LOW_PRIORITY来降低INSERT语句的优先级，这个关键字的作用也适用于UPDATE
和DELETE语句。

```
INSERT LOW_PRIORITY INTO
student(ID, name, dept_name, tot_cred)
VALUES
('00001', 'Hunk', 'Comp. Sci.', 100);
```

3.2 UPDATE语句

```
UPDATE student
SET dept_name = 'Physics'
WHERE name = 'Hunk'; #更新列值

```

```
UPDATE student
SET dept_name = NULL
WHERE name = 'Hunk'; #删除列值

UPDATE IGNORE student
 SET tot_cred = 10; #批量更新时，如果某一行更新失败，整个UPDATE操作被取消
 #加上IGNORE后，即使发生错误，也继续更新。
```

3.3 DELETE语句

```
DELETE FROM student
WHERE name = 'Hunk'; #过滤删除行

DELETE FROM student; #删除所有行，但不删除表本身

TRUNCATE TABLE <table> #如果要删除所有行，这条命令的执行效率更高
#TRUNCATE实际上是删除原来的表并重新创建一个表，而不是逐行删除记录
```

4 创建和操纵表

4.1 创建表

用如下SQL创建takes表：

```
CREATE TABLE takes
 (ID varchar(5),
 course_id varchar(8),
 sec_id varchar(8),
 semester varchar(6),
 year numeric(4,0),
 grade varchar(2),
 PRIMARY KEY (ID, course_id, sec_id, semester, year), #指定主键，由多个列组成
 FOREIGN KEY (course_id,sec_id, semester, year) REFERENCES section(course_id,sec_id, semester, year)
  ON DELETE cascade,
  FOREIGN KEY (ID) REFERENCES student(ID)
  ON DELETE cascade
  );
```

用SHOW CREATE TABLE takes;可以查看MySQL创建这个表时的完整SQL语句，包括了很多自动添加的默认选项：

```
CREATE TABLE `takes` (
```

```
  `ID` varchar(5) NOT NULL,
  `course_id` varchar(8) NOT NULL,
  `sec_id` varchar(8) NOT NULL,
  `semester` varchar(6) NOT NULL,
  `year` decimal(4,0) NOT NULL, #因为前面这几个值被指定为primary key，因此默认设置为NOT NULL
  `grade` varchar(2) DEFAULT NULL, #非primary key的列值默认允许NULL值，除非加上NOT NULL
  PRIMARY KEY (`ID`,`course_id`,`sec_id`,`semester`,`year`),
  KEY `course_id` (`course_id`,`sec_id`,`semester`,`year`),
   CONSTRAINT `takes_ibfk_1` FOREIGN KEY (`course_id`, `sec_id`, `semester`, `year`)
   REFERENCES `section` (`course_id`, `sec_id`, `semester`, `year`)
   ON DELETE CASCADE,
   CONSTRAINT `takes_ibfk_2` FOREIGN KEY (`ID`)
   REFERENCES `student` (`id`)
   ON DELETE CASCADE
 ) ENGINE=InnoDB #默认引擎为InnoDB
DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Notes:

不要把NULL值和空值串混淆。NULL值是没有值，不是空串。空串是一个有效值，不是无值。

AUTO_INCERMENT和DEFAULT

```
CREATE TABLE test
 (ID int AUTO_INCREMENT, #每插入一个新列时，ID从1开始自增长
 name varchar(8) NOT NULL DEFAULT 'test', #insert时没有值默认设置为test,不允许函数作为默认值
 PRIMARY KEY(ID)
 );
```

Notes:

每个表只允许一个AUTO_INCERMENT列，而且它必须被索引，比如通过使它成为主键。

如果一个列被指定为AUTO_INCREMENT，insert时可以指定一个值，只要它是唯一的（至今未使用过）即可，

该值将被用于替代自动生成的值，后续的增量将开始使用该手工插入的值。

如果在表A插入新列时，在关联表B中很可能也需要插入新列，而且需要表A中新列自增长的值，

这种情况可以用last_inert_id()函数来获取

```
SELECT last_insert_id() #此语句返回最后一个AUTO_INCREMENT的值，然后可以用于后续SQL语句
```

ENGINE

InnoDB:可靠的事务处理引擎，不支持全文本搜索。

MEMORY:功能等同于MyISAM，数据存储在内存而不是磁盘中，速度很快，比较适合临时表。

MyISAM：一个性能极高的引擎，支持全文本搜索，但不支持事务处理。

外键不能跨引擎：使用一个引擎的表不能引用具有使用不同引擎的表的外键。

4.2 更新表结构

```
ALTER TABLE takes
ADD test varchar(8); #添加列

ALTER TABLE takes
DROP COLUMN test; #添加列
```

ALTER TABLE的常见用途是定义外键

```
ALTER TABLE orders
ADD CONSTRAINT fk_orders_customers FOREIGN KEY (cust_id)
REFERENCES customers (cust_id);
```

复杂的表结构更改一般需要手动删除过程，设计步骤如下：

1. 用新的列布局创建一个新表；
2. 使用INSERT SELECT语句从旧表赋值数据到新表。如有必要，可以使用转换函数和计算字段。
3. 检验包含所需数据的新表。
4. 重命名旧表或者删除。
5. 用旧表原来的名字重命名新表。
6. 根据需要，重新创建触发器、存储过程、索引和外键。

Notes:

在使用ALTER TABLE之前最好做一个完整的备份，包括模式和数据。

数据表的更改不能撤销，如果增加了不需要的列，可能无法删除；

如果删除了不应该删除的列，可能会丢失该列中的数据。

4.3 删除表

```
DROP TABLE takes;
```

4.4 重命名表

```
RENAME TABLE takes TO takes_new,
 student TO student_new;
```

5 视图

视图为虚拟的表，它们包含的并不是数据，而是根据需要检索数据的查询。视图提供了一种MySQL的SELECT语句
层次的封装，可以用来简化数据处理以及重新格式化基础数据或者保护基础数据。
每次使用视图时，都必须处理查询执行时所需的任一个检索。
比如下面的联结表查询：

```
SELECT takes.ID, course_id, name, student.dept_name, building
FROM takes, student, department
WHERE takes.ID = student.ID
AND student.dept_name = department.dept_name
AND building = 'Taylor' ;
```

我们可以先创建可以重复使用的视图tsd：

```
CREATE VIEW tsd AS
SELECT takes.ID, course_id, name, student.dept_name, building
FROM takes, student, department
WHERE takes.ID = student.ID
AND student.dept_name = department.dept_name;
```

然后基于视图执行查询：

```
SELECT * FROM tsd WHERE building = 'Taylor';
```

利用视图，可以一次性编写基础的SQL，然后根据需要多次使用。

Notes:
一般的，应该将视图用于检索（SELECT），而不用于更新（INSERT, UPDATE, DELETE）。

6 存储过程

6.1 为什么需要存储过程

https://blog.csdn.net/u012299594/article/details/84476055

6.2 创建存储过程

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `pcdAvgCredForOneDept`(
 IN dept VARCHAR(15),
 OUT avg_cred DECIMAL(8, 2)
)
 COMMENT 'Compute avg cred'
BEGIN
 -- Declare varibale for avg
 DECLARE avg_tmp DECIMAL(8,2);

 -- Get the avg including param 'dept'
 IF dept = 'ALL' THEN
 -- Get avg cred of all departments
 SELECT AVG(tot_cred) FROM student INTO avg_tmp;
 ELSE
 -- Get avg cred of a special department
 SELECT AVG(tot_cred) FROM student WHERE dept_name = dept INTO avg_tmp;
 END IF;

 -- And finnal, save to out varibale
 SELECT avg_tmp INTO avg_cred;

END
```

6.3 使用存储过程

```
mysql> CALL pcdAvgCredForOneDept('ALL', @avg_cred);
Query OK, 1 row affected (0.01 sec)

mysql> SELECT @avg_cred;
+-----------+
| @avg_cred |
+-----------+
| 75.29 |
+-----------+
 mysql> CALL pcdAvgCredForOneDept('Physics', @avg_cred);
 Query OK, 1 row affected (0.00 sec)

 mysql> SELECT @avg_cred;
 +-----------+
 | @avg_cred |
```

```
+-----------+
| 34.00 |
+-----------+
```

6.4 删除存储过程

```
DROP PROCEDURE pcdAvgCredForOneDept IF EXISTS;
```

Notes:

1. 所有MySQL变量都以@开始。
2. 不能通过一个参数返回多个行和列。
3. MySQL将编写存储过程的安全访问和执行存储过程的安全和访问区分开来。

7 游标

使用SELECT语句查询时返回的是结果集，而有时候我们需要逐行（或者多行）的处理数据，比如在交互式应用中，
用户需要滚动屏幕上的数据，并对数据进行浏览或修改。
游标是一个存储在MySQL服务器上的数据库查询，它不是SELECT语句，而是SELECT语句检索出来的结果。
MySQL游标只能用于存储过程。

使用游标：

1. 在使用游标之前必须先用DECLARE定义，这个过程实际上没有检索数据，只是定义要使用的SELECT语句。
2. 用OPEN打开游标后可以使用，这个过程用前面定义的SELECT语句把数据检索出来。
3. 对于填有数据的游标，根据需要使用FETCH来访问每一行结果。
4. 在游标使用结束后，必须用CLOSE关闭游标。CLOSE释放游标使用的所有内部内存和资源，

因此在每个游标不再需要时都应该关闭。如果不明确关闭游标，MySQL将会在到达END语句时自动关闭它。

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `pcdAvgCredForEveryDept`()
BEGIN
 -- Declare local variables
 DECLARE done BOOLEAN DEFAULT 0;
 DECLARE o VARCHAR(15);
```

```
 DECLARE t DECIMAL(8, 2);

 -- Declare the cursor
 DECLARE deptcursor CURSOR
  FOR
  SELECT dept_name FROM department;

  -- Declare continue handler
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done=1;

  -- Create a table to store the results
  CREATE TABLE IF NOT EXISTS avgcredtotals
  (dept VARCHAR(15), avg_cred DECIMAL(8, 2));

  -- Clear the old data in table
  DELETE FROM avgcredtotals;

  -- Open the cursor
  OPEN deptcursor;

  -- Loop through all rows
  REPEAT
  -- Get one department
  FETCH deptcursor INTO o;

  -- Get the avg cred for this department
  CALL pcdAvgCredForOneDept(o, t);

  -- Insert department and avg cred into avgcredtotals
  INSERT INTO avgcredtotals(dept, avg_cred)
  VALUES(o, t);

  -- End of loop
  UNTIL done END REPEAT;

  -- Close the cursor
  CLOSE deptcursor;
END
```

```
mysql> CALL pcdAvgCredForEveryDept;
Query OK, 1 row affected (0.07 sec)
```

```
mysql> select * from avgcredtotals;
+------------+----------+
| dept | avg_cred |
+------------+----------+
| Biology | 120.00 |
| Comp. Sci. | 89.20 |
| Elec. Eng. | 79.00 |
| Finance | 110.00 |
| History | 80.00 |
| Music | 38.00 |
| Physics | 34.00 |
| Physics | 34.00 |
+------------+----------+
```

Notes:

1. 这里使用了一个CONTINUE HANDLER，它是在条件出现时被执行的代码。
2. SQLSTATE '02000'是一个未找到条件，当FETCH访问到结果集的末尾没有更多行访问时，该条件出现。

8 触发器

8.1 概述

触发器机制就类似于我们在编程中常用到的异步处理模型：针对某种事件注册对应handler，事件被触发时，
自动调用相应的handler来进行梳理。

MySQL触发器支持的事件包括6个：

```
AFTER INSERT
BEFORE INSERT
AFTER DELETE
BEFORE DELETE
AFTER UPDATE
BEFORE UPDATE
```

创建触发器

```
CREATE TRIGGER newTrigger AFTER INSERT ON student
FOR EACH ROW SELECT 'name'; #插入时查询输出新行里的name
```

删除触发器

```
DROP TRIGGER newTrigger;
```

Notes:

1. 触发器一定和某张表关联，也只有表才支持触发器，视图和临时表不支持。
2. 触发器名必须在每个表中唯一，但不是在数据库中唯一。
3. 每个表每个事件只允许一个触发器，所以每个表最多支持6个触发器。
4. 如果BEFORE触发器失败，MySQL不支持请求的操作。
5. 如果BEFORE触发器或者SQL语句本身失败，AFTER触发器将不被执行。

8.2 INSERT触发器

```
CREATE TRIGGER insertTrigger AFTER INSERT ON student
FOR EACH ROW SELECT NEW.name;
```

Notes:

1. 在INSERT触发器代码内，可引用一个名为NEW的虚拟表，访问被插入的行。
2. 在BEFORE INSERT触发器中，NEW中的值也可以被更新（允许更改被插入的值）。
3. 对于AUTO_INCERMENT列，NEW在INSERT执行之前包含0，INSERT执行之后包含自动生成的值。
4. 对于INSERT和UPDATE，BEFORE常用于数据验证和净化（保证插入或更新的数据确实是需要的数据）。

8.3 DELETE触发器

在下面的触发器中，删除数据之前先将其归档到其它存档表中。

```
CREATE TRIGGER deleteTrigger BEFORE DELETE ON student
FOR EACH ROW
BEGIN
 INSERT INTO archive_student(name, dept_name)
 VALUES(OLD.name, OLD.dept_name);
END;
```

Notes:

1. 在DELETE触发器代码内，可引用一个名为OLD的虚拟表，访问被删除的行。
2. OLD中的值全都是只读的，不能更新。
3. 使用BEGIN END可以容纳多条SQL语句，在BEGIN END块中一条挨着一条。

8.3 UPDATE触发器

```
CREATE TRIGGER updateTrigger BEFORE UPDATE ON student
FOR EACH ROW SET NEW.name = Upper(NEW.name); #保证名字总是大写
```

Notes:

1. 在UPDATE触发器代码内，可引用一个名为OLD的虚拟表访问旧值，用NEW表访问新值。
2. 在BEFORE UPDATE触发器中，NEW中的值也可以被更新（允许更改新值）。
3. OLD表中的值都是只读的，不能更新。

关于触发器的其它说明

1. 创建触发器可能需要特殊的安全访问权限，但是触发器的执行时自动的。
2. 应该用触发器来保证数据一致性（大小写，格式等）。在触发器中执行这类型处理的优点是透明进行，与客户机无关。
3. 触发器的一种很有意义的使用时创建审计跟踪，比如把更改记录到另一个表。
4. MySQL触发器不支持CALL语句，意味着不能从触发器内调用存储过程，所需代码需要复制到触发器内。

9 事务处理

事务其实就是一组SQL语句。

事务处理可以用来维护数据库的完整性，保证成批的MySQL操作要么完全执行，要么完全不执行。

事务处理的精髓就在于rollback，在某条SQL语句执行失败时或者系统故障重启后，通过rollback让数据可以保持一致性。

在批量SQL中使用事务处理：

```
SELECT * FROM student;
START TRANSACTION
SAVEPOINT deletel; #创建保留点
DELETE FROM takes;
SELECT * FROM student;
ROLLBACK TO deletel; #这里的ROLLBACK没什么实际意义
DELETE FROM course;
SELECT * FROM student;
COMMIT;
```

Notes:

1. ROLLBACK只能在一个事务处理内（START TRANSACTION之后）使用。
2. 事务处理用来管理INSERT，UPDATE和DELETE。DROP，CREATE操作无法回退。

3. 一般的MySQL语句都是直接针对数据库表执行和编写的，提交（写或保存）自动完成，即隐含提交。
4. 事务处理块中，提交不会自动进行，需要使用COMMIT语句显示提交。
5. 通过保留点可以指定ROLLBACK到什么位置，保留点在事务完成后自动释放。

修改默认地自动提交行为：

```
SET autocommit=0; #不自动提交
SET autocommit=1; #自动提交
```

10 安全管理

Notes：

尽量不要在root账户下直接操作数据库。

不推荐直接往user表里插入记录来创建新用户。

10.1 用户账户管理

```
mysql> USE mysql;
mysql> SELECT user FROM user;
+------------------+
| user |
+------------------+
| mysql.infoschema |
| mysql.session |
| mysql.sys |
| root |
+------------------+
```

创建新用户

```
CREATE USER hunk IDENTIFIED BY 'P@ssw0rd';
```

重命名用户账号

```
RENAME USER hunk TO hunk1;
```

删除用户账号

```
DROP USER hunk1;
```

更改口令

```
SET PASSWORD FOR hunk1 = Password('new pass');
```

新口令必须传递到Password()函数加密。

```
SET PASSWORD = Password('new pass'); #更改当前用户口令
```

10.2 用户权限管理

查询初始访问权限

```
mysql> SHOW GRANTS FOR hunk1;
+-----------------------------------+
| Grants for hunk1@% |
+-----------------------------------+
| GRANT USAGE ON *.* TO `hunk1`@`%` |
+-----------------------------------+
```

这里只有一个权限USAGE ON *.*，USAGE表示无访问权限，所以就是根本没有任何权限。MySQL的权限用用户名和主机名结合定义。如果不指定主机名，则使用默认地主机名%（授予用户访问权限而不管主机名）。

授予新的权限

```
GRANT SELECT,INSERT ON hunkdb.* TO hunk1;
```

这条GRANT允许用户在hunkdb.*（hunkdb数据库所有表）上使用SELECT和INSERT操作。

每个GRANT添加用户的一个权限，mys6duqu所有授权，并根据他们确定权限。

撤销用户权限

```
REVOKE SELECT ON hunkdb.* TO hunk1;
```

被撤销的访问全新必须存在，否则会出错。

在使用GRANT和REVOKE时，用户账号必须存在，但对所涉及的对象没有这个要求。因此，允许在创建数据库和表之前设计和实现安全措施。

GRANT和REVOKE可在几个层次上控制访问权限：

1. 整个服务器，使用GRANT ALL和REVOKE ALL;
2. 整个数据库，使用ON <database>.*;
3. 特定的表，使用ON <database>.<table>;
4. 特定的列。
5. 特定的存储过程。

可以授予或撤销的权限列表

。。。

11 数据库维护

11.1 备份数据

可能的备份解决方案包括：

1. mysqldump --转储所有数据库内容到某个外部文件。
2. mysqlhotcopy --从一个数据库复制所有数据，但并非所有引擎都支持。
3. BACKUP TABLE或者SELECT INTO OUTFILE --转储所有数据到某个外部文件，数据可用RESTORE TABLE来还原。

为了保证所有数据被写到磁盘（包括索引数据），可能需要在进行备份前使用FLUSH TABLES语句。

11.2 数据库维护

ANALYZE TABLE语句用来检查表键是否正确。

```
mysql> ANALYZE TABLE student;
+----------------+---------+----------+----------+
| Table | Op | Msg_type | Msg_text |
+----------------+---------+----------+----------+
| hunkdb.student | analyze | status | OK |
+----------------+---------+----------+----------+
```

CHECK TABLE用来针对许多问题对表进行检查，在MyISAM表上还对索引检查。

CHECK TABLE支持一系列用于MyISAM表的检查：

1. CHANGED检查自最后一次检查以来改动过的表。
2. EXTENED执行最彻底的检查。
3. FAST只检查为正常关闭的表。
4. MEDIUM检查所有被删除的链接并进行键检验。
5. QUIK只进行快速扫描。

如果MyISAM表访问产生不正确或者不一致的结果，可能需要REPAIR TABLE来修复相应的表。

这条语句不应该经常使用，否则会有更大的问题需要解决。

如果从一个表中删除大量数据，应该使用OPTIMIZE TABLE来收回所用空间，从而优化表的性能。

11.3 诊断启动问题

在排除系统启动问题时，首先应该尽量用手动启动服务器。
MySQL服务器自身通过mysqld命令启动。

```
mysqld --help
mysqld --verbose --help #查看非常详细的启动参数
```

11.4 查看日志文件

主要日志

1. 错误日志 --包含启动和关闭问题以及任意关键错误的细节，名字通常为hostname.err，可用--log-err更改。
2. 查询日志 --记录所有MySQL活动，名字通常为hostname.log，名字可用--log更改。
3. 二进制日志 --记录更新过数据的所有语句，日志名字通常为binlog.xxxx，名字可用--log-bin更改。

```
SHOW MASTER STATUS;
SHOW MASTER LOGS; #查看当前有哪些log
```

这些日志一般都位于/etc/my.cnf中datadir配置的目录。

查看二进制日志时最好先重定向到文件：

```
mysqlbinlog /var/lib/mysql/binlog.000001 > bin.log
```

关于日志配置

日志的一些配置都在/etc/my.cnf文件mysqld域：

```
cat /etc/my.cnf
[mysqld]
#
datadir=/var/lib/mysql
socket=/var/lib/mysql/mysql.sock

log-error=/var/log/mysqld.log.err #错误日志
log=/var/log/mysqld.log #查询日志
log-bin=/var/log/mysqld.log.bin #二进制日志

 pid-file=/var/run/mysqld/mysqld.pid
```

在使用日志时，可用FLUSH LOGS语句来刷新和重新开始所有日志文件。

12 改善性能

关于改善性能的一些讨论点：

1. MySQL具有特定的硬件建议，关键的生产DBMS应该运行在专用的服务器上。
2. MySQL用一系列默认设置预先配置，但过一段时间后需要调整内存分配、缓冲区大小等，为查看当前配置，
   可使用SHOW VARIABLES;和SHOW STATUS;
3. MySQL属于多用户多线程DBMS，如果遇到显著性能不良，可使用SHOW PROCESSLIST;
   显示所有活动进程极其线程ID和执行时间，必要时可以用KILL终结特定进程。
4. SELECT语句总有多种写法，应试验联结、并、子查询等，找出最佳方法。
5. 一般的，存储过程执行得比一条一条执行其中的各条MySQL语句快。
6. 使用正确地数据类型。
7. 绝不要检索比需求还多的数据，比如SELECT *（除非真正需要所有列）。
8. 在导入数据时，应该关闭自动提交。可能还想删除索引（包括FULLTEXT索引），然后在导入完成后再重建它们。
9. 必须索引数据库表以改善数据库检索性能。
10. 如果SELECT中有很多OR条件，使用多条SELECT语句和连接它们的UNION语句，能看到极大的性能改进。
11. LIKE很慢，一般最好使用FULLTEXT 而不是LIKE。
12. 数据库是不断变化的实体，所以表的优化和配置是需要持续去做的。
13. 最重要的规则，每条规则在某些条件下都会被打破。