

MySQL 数据库学习笔记

郑华

2018 年 12 月 12 日

第一章 基础概念

1.1 术语

- **数据库:** 数据库是一些关联表的集合。
- **数据表:** 表是数据的矩阵。在一个数据库中的表看起来像一个简单的电子表格。
- **列:** 一列 (数据元素) 包含了相同的数据, 例如邮政编码的数据。
- **行:** 一行 (= 元组, 或记录) 是一组相关的数据, 例如一条用户订阅的数据。
- **冗余:** 存储两倍数据, 冗余降低了性能, 但提高了数据的安全性。
- **主键:** 主键是唯一的。一个数据表中只能包含一个主键。你可以使用主键来查询数据。
- **外键:** 外键用于关联两个表。
- **复合键:** 复合键 (组合键) 将多个列作为一个索引键, 一般用于复合索引。
- **索引:** 使用索引可快速访问数据库表中的特定信息。索引是对数据库表中一列或多列的值进行排序的一种结构。类似于书籍的目录。
- **参照完整性:** 参照的完整性要求关系中不允许引用不存在的实体。与实体完整性是关系模型必须满足的完整性约束条件, 目的是保证数据的一致性。

1.2 教程集合地址

<https://blog.csdn.net/orangleliu/article/details/54694272>

第二章 数据库基本操作

2.1 SQL 分类

- 数据库查询：代表关键字 `select`
- 数据库操纵：代表关键字 `insert delete update`
- 数据库定义：代表关键字 `create drop alter`
- 事务控制：代表关键字 `commit rollback`
- 权限控制：代表关键字 `grant revoke`

2.2 常用命令

表 2.1: 常用命令

类型	命令
显示当前的数据库们	<code>show databases;</code>
使用某个数据库	<code>use databaseName;</code>
显示数据库中的表们	<code>show tables;</code>
查看表的创建语句	<code>show create table tableName;</code>
查看表的结构	<code>desc tableName;</code>
重命名 (列名、表明)	<code>as , 如 select lower(ename) as E from emp;</code>
创建数据库	<code>create database Name;</code>
设置字符集	<code>set NAMES 'utf8'; SET character_set_xx = utf8;</code>
终止一条语句	<code>\c</code>

2.3 MySql 框架、执行流程

<https://www.cnblogs.com/annsshadow/p/5037667.html>

2.3.1 MySQL 框架

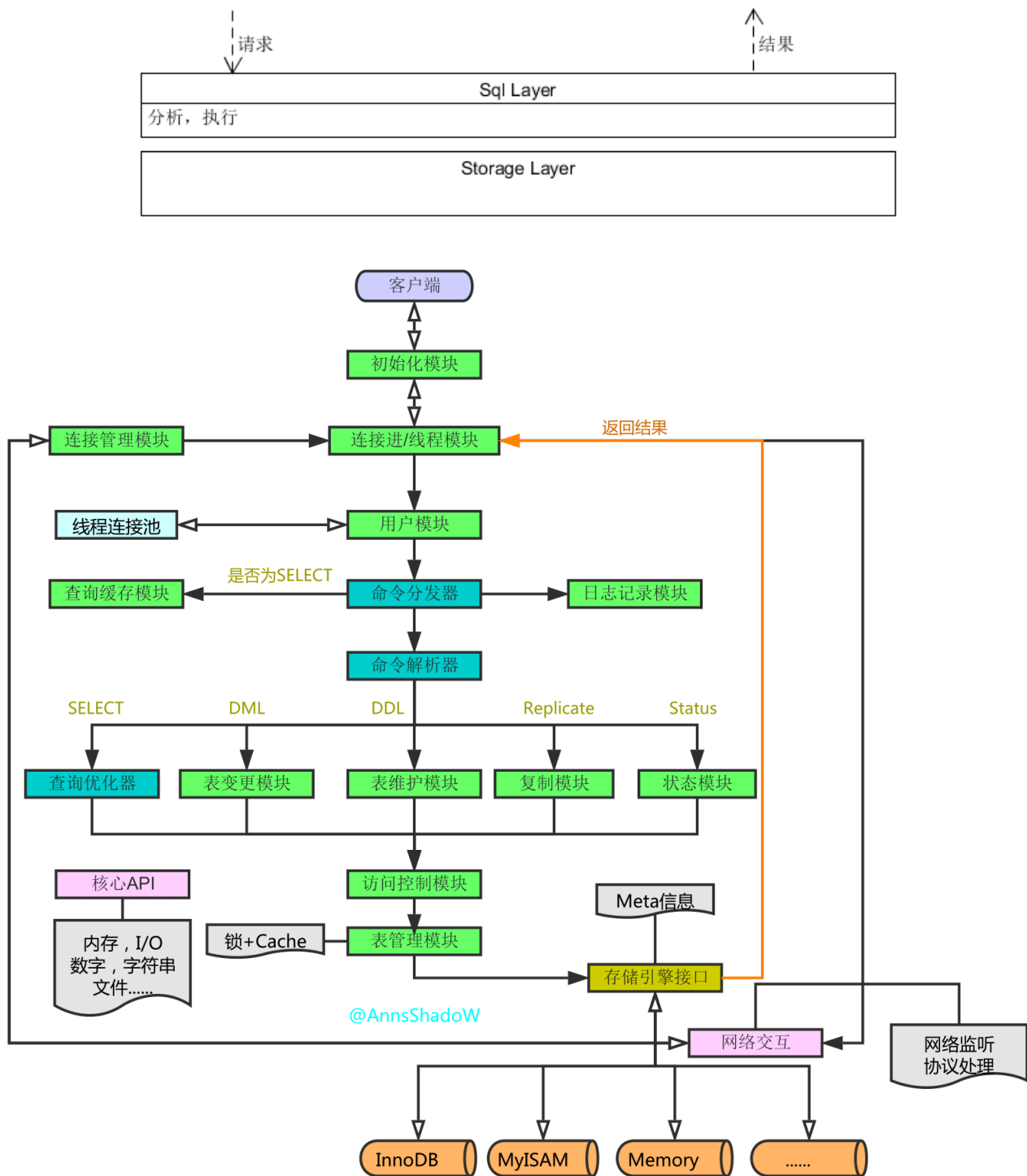


图 2.1: MySQL 框架

2.3.2 查询执行流程

连接

1. 客户端发起一条 Query 请求，监听客户端的‘连接管理模块’接收请求

2. 将请求转发到‘连接进/线程模块’
3. 调用‘用户模块’来进行授权检查
4. 通过检查后，‘连接进/线程模块’从‘线程连接池’中取出空闲的被缓存的连接线程和客户端请求对接，如果失败则创建一个新的连接请求

处理

1. 先查询缓存，检查 Query 语句是否完全匹配，接着再检查是否具有权限，都成功则直接取数据返回
2. 上一步有失败则转交给‘命令解析器’，经过词法分析，语法分析后生成解析树
3. 接下来是预处理阶段，处理解析器无法解决的语义，检查权限等，生成新的解析树
4. 再转交给对应的模块处理
5. 如果是SELECT 查询还会经由‘查询优化器’做大量的优化，生成执行计划
6. 模块收到请求后，通过‘访问控制模块’检查所连接的用户是否有访问目标表和目标字段的权限
7. 有则调用‘表管理模块’，先是查看 table cache 中是否存在，有则直接对应的表和获取锁，否则重新打开表文件
8. 根据表的 meta 数据，获取表的存储引擎类型等信息，通过接口调用对应的存储引擎处理
9. 上述过程中产生数据变化的时候，若打开日志功能，则会记录到相应二进制日志文件中

结果

1. Query 请求完成后，将结果集返回给‘连接进/线程模块’
2. 返回的也可以是相应的状态标识，如成功或失败等
3. ‘连接进/线程模块’进行后续的清理工作，并继续等待请求或断开与客户端的连接

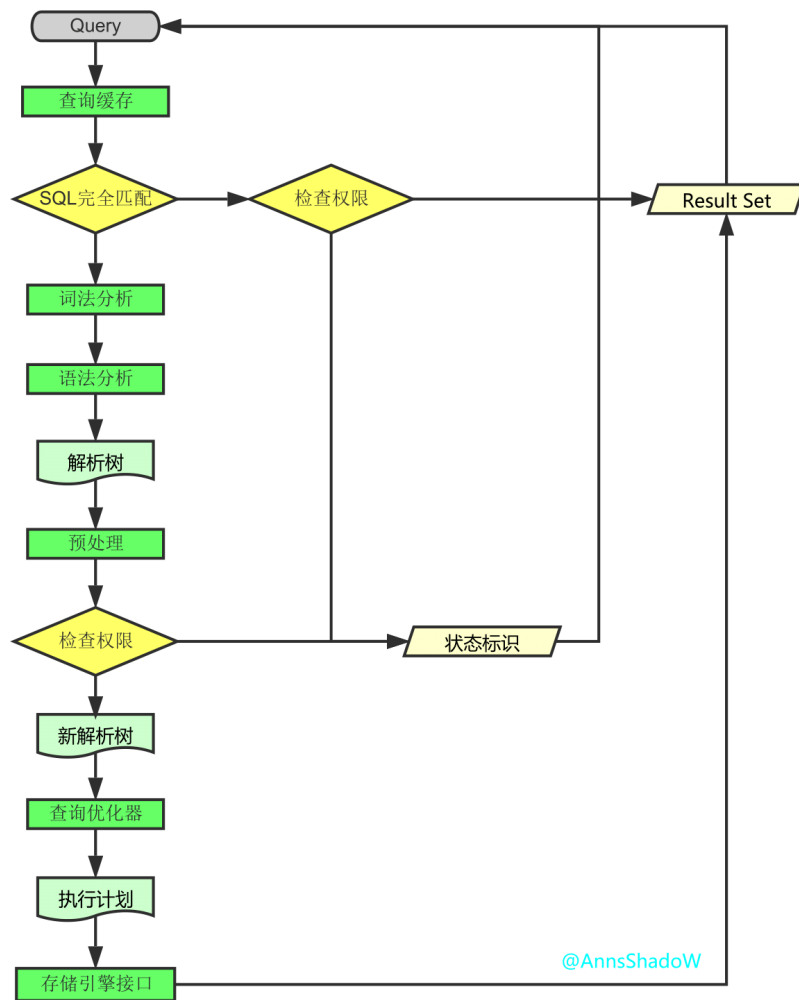


图 2.2: 查询执行流程

第三章 查询

3.1 基本查询语句

3.1.1 条件查询

表 3.1: 查询符号

运算符	功能说明
=	等于
!=	不等于
between ... and ..	等同于 >= ... and <= ...
is null	为null(is not null 不为空)
and	并且
or	或者
in	包含, 相当于多个or,(not in 不在这个范围内)
not	取非
like	为模糊查询, 支持% 或_ 匹配, 其中% 匹配任意个字符, _ 只匹配一个字符

example ->

```
// 执行顺序
select // 3
    xx, xx2, xx3
from // 1
    XX
where // 2
    xx = xx;

// in 示例 查找job是什么的, 不是什么的
select
    ename, job
from
    emp
```

```

where
    job in('MANAGER','SALESMAN');

select
    ename,job
from
    emp
where
    job not in('MANAGER','SALESMAN');

// like 示例, 查找以S 开头的名字
select
    ename
from
    emp
where
    ename like 'S%'

```

3.1.2 排序

order by

```

// order 示例 默认升序, (desc 降序)
select
    ename,salary
from
    emp
order by
    salary

// 按照第几个字段排序
select
    ename,salary // 1,2 字段
from
    emp
order by
    2 // 第2个字段

// 多个字段排序, ename 升序, salary 降序, 使用逗号分割
select
    ename,salary
from
    emp
order by
    salary desc, ename

```

3.1.3 数据处理函数（单行）

处理单行后结束

- `lower` : 转换小写
- `upper` : 转换大写
- `substr` : 取子串（被截取的串，起始位置，截取长度）
- `length` : 取长度
- `trim` : 去空格
- `round` : 四舍五入
- `rand()` : 生成随机数
- `ifnull(xx, num)`: 可以将`null` 值转换成一个具体值

3.1.4 分组函数、聚合函数（多行）

处理多行后结束，自动忽略空值

先分组，然后再执行分组函数，而 `where` 在分组函数之前执行，所以不能 `where` 中不能出现分组函数

- `count` : 取得记录数
- `sum` : 求和
- `avg` : 求平均
- `max` : 取最大值
- `min` : 取最小值

`distinct` 去重关键字 -> `select distinct job from emp`; 只能出现在所有字段的最前面
`select count(distinct job) from emp`;

3.2 分组查询

`group by` : 通过哪个或哪些字段进行分组，使用后 `select` 后只能跟参与分组的字段和分组函数。

example-> 找出每个工作岗位的最高薪水【先按照工作岗位分组，使用 `max` 函数求每一组的最高工资】

```
// 先按照job 分组，然后对每一组使用max(salary) 求最大值。
```

```
select    //3
    max(salary)
from      //2
    emp;
group by  //1
    job;
```

```
// 结合where 限定分组前条件，即分组前过滤
```

```
select
    job, max(sal)
from
    emp
where
    job != 'MANAGER'
group by
    job;
```

example-> 找出每个工作岗位的平均薪水，要求显示平均薪水大于 1500 where 处理不了

having 与 **where** 都是为了完成数据的过滤，**where** 和 **having** 后面都是添加过滤条件，**where** 是在 **group by** 之前执行，而 **having** 是在 **group by** 后执行。

```
//上例子解法
```

```
select
    job, avg(sal)
from
    emp
group by
    job
having
    avg(sal) > 1500;
```

3.2.1 查询语句总结

关键字顺序不能变：

```
select
    ...
from
    ...
where
    ...
group by
    ...
having
    ...
order by
```

|| ...

执行顺序：

1. `from` 从某张表中检索数据
2. `where` 经过某条件进行过滤
3. `group by` 然后分组
4. `having` 分组之后不满意再过滤
5. `select` 查询出来
6. `order by` 排序输出

3.3 连接查询

查询的时候只从一张表检索数据称为单表查询

在实际的开发中，数据并不是存储在一张表中的，是同时存储在多张表中，这些表和表之间存在关系，我们在检索的时候通常需要将多张表联合起来取得有效数据，这种多表查询被称为连接查询或者叫做跨表查询。

连接查询根据连接方式可以分为如下方式：

- 内连接
 - 等值连接
 - 非等值连接
 - 自连接
- 外连接
 - 左外连接
 - 右外连接
- 全连接【几乎不用】

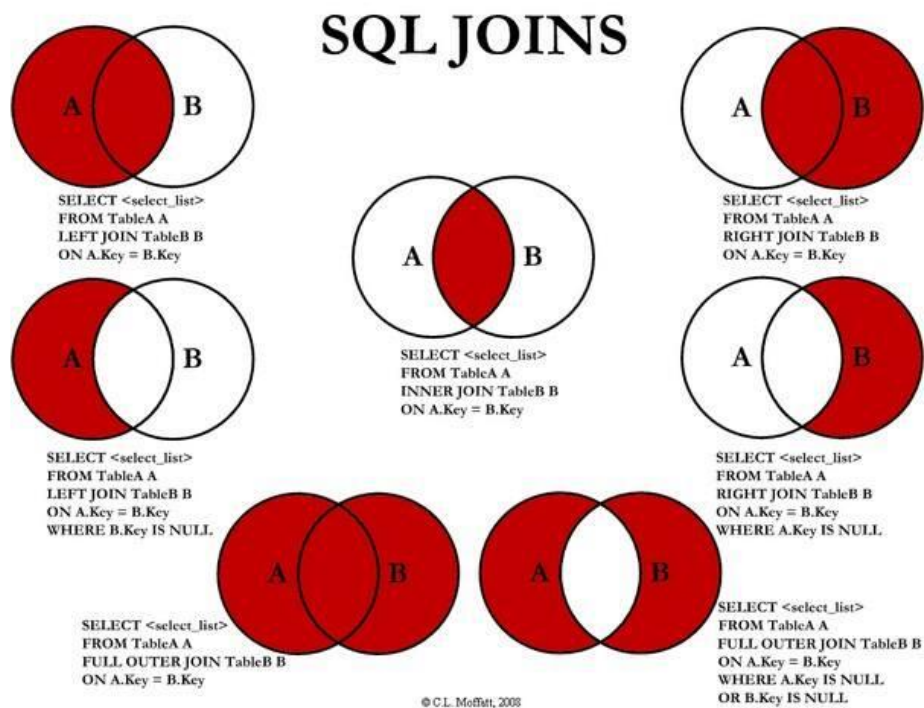


图 3.1: SQL Joins

3.3.1 内连接

查找两张表匹配的数据。

A 表和 B 表能够完全匹配的记录查询出来，被称为内连接。

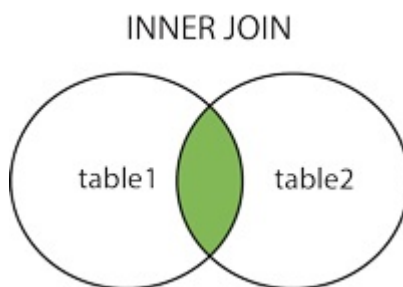


图 3.2: 内连接

别名的使用，内连接的等值连接 在进行多表连接查询的时候，尽量给表起别名，这样效率高，可读性高

```
// 将表emp 用别名 e表示..
// 查询员工名与其对应的部门名
select
    e.ename, d.dname
from
    emp e, dept d;
where
    e.deptno = d.deptno
```

// SQL99 语法,使得表连接独立出来了, 结构更清晰

```
select
    e.ename, d.dname
from
    emp e
join // 内连接的inner 可以省略
    dept d
on
    e.depno = d.depno;
```

内连接的非等值连接 范围

// 找出员工名, 薪水, 与他的薪水等级

```
select
    e.name, e.sal, s.grade
from
    emp e
join
    salgrade s
on e.sal >= s.lower and e.sal <= s.higher; // 可以使用between and 替代
```

内连接的自连接 自己与自己连接, 将自己视为两张表

// 找出每一个员工的上级领导, 要求显示员工名以及对应的领导名

表结构:

```
empno ename mgr
7369 SMITH 7123
```

// 要点: 将自己视为两张表

```
select
    a.ename empname, b.ename leaderName
from
    emp a
join
    emp b
on
    a.mgr = b.empno;
```

3.3.2 外连接

A 表和 B 表能够匹配的记录查询出来之外, 将其中一张表的记录完全无条件的完全查询出来, 对方表没有匹配的记录, 会自动模拟出 *NULL* 与之匹配。

外连接查询的结构条数 \geq 内连接的查询结果数量

可以添加除了内连接外的其他数据。

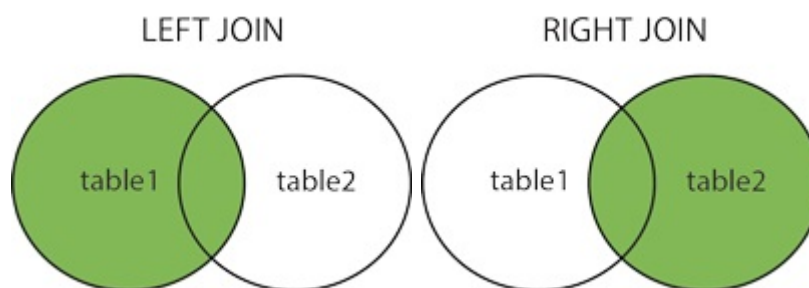


图 3.3: 外连接 (左右)

example -> 找出每一个员工对应的部门名称，并且显示所有部门名称，注意部门可能没有员工。

左外连接 `select e.ename, d.dname from dept d left join emp e on e.deptno = d.deptno;`

右外连接 `select e.ename, d.dname from emp e right join dept d on e.deptno = d.deptno; //`
outer 省略

总结 希望将哪边表的数据完全显示出来，join 的前边的修饰词 right left 可以恰好说明，如上，希望将 dept 表完全显示，那么先写dept 的话，那么就在join 的左边，就是 left join.

3.3.3 内连接外连接原理性能分析

<https://www.cnblogs.com/cdf-opensource-007/p/6507678.html>

<https://www.cnblogs.com/cdf-opensource-007/p/6517627.html>

3.4 子查询

<https://www.cnblogs.com/zhuiluoyu/p/5822481.html>

3.5 union

UNION 操作符用于合并两个或多个 SELECT 语句的结果集

```
SELECT column_name(s) FROM table_name1 // 如只有 1
UNION
SELECT column_name(s) FROM table_name2 // 如只有 2

// 则结果为
1
2
```

示例: http://www.w3school.com.cn/sql/sql_union.asp

3.6 limit

用来获取一张表中的某部分数据，只在 MySQL 数据特有的。

```
// 找到员工表中前5条记录
select ename
from emp
limit 5; //从下标0开始

select ename
from emp
limit 0,5; // 从0下标开始，查找前5条

// 找到工资在3到9名的员工
select salary
form emp
order by salary desc
limit 2,7; // 第三个的下标为2，一共7条
```

3.7 执行顺序

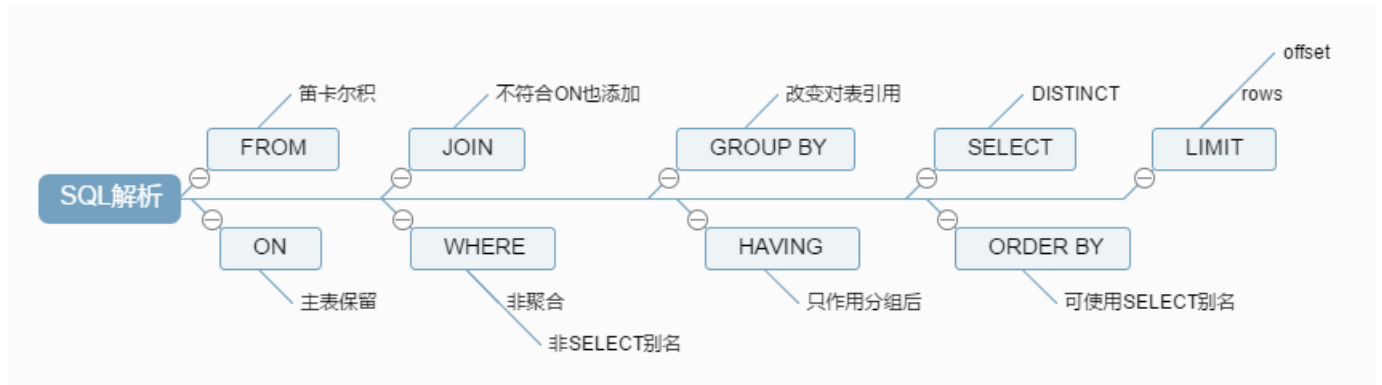


图 3.4: SQL 解析顺序

```
FROM <left_table>
ON <join_condition>
<join_type> JOIN <right_table>
WHERE <where_condition>
GROUP BY <group_by_list>
HAVING <having_condition>
SELECT
DISTINCT <select_list>
ORDER BY <order_by_condition>
LIMIT <limit_number>
```

3.8 执行过程

<https://www.cnblogs.com/cdf-opensource-007/p/6502556.html>

1. 加载数据表至内存：一条查询的 sql 语句先执行的是 FROM student 负责把数据库的表文件加载到内存中去。（mysql 数据库在计算机上也是一个进程，cpu 会给该进程分配一块内存空间）
2. 条件过滤：WHERE grade < 60，会把所示表中的数据进行过滤，取出符合条件的记录行，生成一张临时表
3. 分组：GROUP BY `name`会把临时表在内存中切分成若干临时表。
4. 选择：SELECT 的执行读取规则分为 sql 语句中有无 GROUP BY 两种情况。
 - 当没有 GROUP BY 时，SELECT 会根据后面的字段名称对内存中的一张临时表整列读取。
 - 当查询 sql 中有 GROUP BY 时，会对内存中的若干临时表分别执行 SELECT，而且只取各临时表中的第一条记录，然后再形成新的临时表。这就决定了查询 sql 使用 GROUP BY 的场景下，SELECT 后面跟的一般是参与分组的字段和聚合函数，否则查询出的数据要是情况而定。另外聚合函数中的字段可以是表中的任意字段，需要注意的是聚合函数会自动忽略空值。
5. 对分组后的数据再次过滤：HAVING num >= 2 对上图所示临时表中的数据再次过滤，与WHERE语句不同的是HAVING 用在GROUP BY 之后，WHERE 是对FROM student 从数据库表文件加载到内存中的原生数据过滤，而HAVING 是对SELECT 语句执行之后的临时表中的数据过滤。
6. 对以上的临时表进行排序：ORDER BY xx DESC|ASC

第四章 增、删、改

4.1 表、(with 约束)

4.1.1 create table tableName(columnName type(length) [constraints]);

```
create table tableName(  
    columnName dataType(length) constraints,  
    ...  
);  
set character_set_result = 'gbk';  
  
drop table tableName;  
drop table if exist tableName; //MySQL 特色  
  
create table tableName as select * from existTableName; // 根据已创建的表创建新表
```

4.1.2 数据类型

- VARCHAR 可变长度字符串
- CHAR 定长字符串
- INT、BIGINT、FLOAT、DOUBLE 基础数据类型
- DATE 日期类型
- BLOB 2 进制大对象-> 图片
- CLOB 字符大对象-> 比较大的字符串

4.2 表结构

4.2.1 `alter table tableName add newColumnName type(length);`

4.2.2 `alter table tableName modify column newType(length);`

4.2.3 `alter table tableName drop column;`

4.3 数据

4.3.1 `insert into tableName(column,...) values (value1,...);`

4.3.2 `update tableName set columnName=newValue,... where xx;`

当不指定条件时，将全表的该字段全部更新。

4.3.3 `delete from tableName where xx;`

4.4 约束

4.4.1 非空约束 `not null`

不能为空

4.4.2 唯一性约束 `unique`

不能重复但是可以为 NULL，该字段值具有唯一性

列级约束

```
create table tb(  
    ..,  
    column varchar(32) unique,  
    ..);
```

表级约束

```
create table tb(  
    ..,  
    column varchar(32),  
    ..,  
    constraint consName unique(column1,...)  
);
```

当使用表级约束时，表示多个字段联合起来后唯一即可。而表级约束可以有名称是为了以后方便删除该约束。

4.4.3 主键约束 primary key

此列必须是**唯一并且非空**

每个表都应该有一个主键，并且每个表只能有一个主键。但是注意，并不是说该主键只能在一列上作用，它具有表级约束的联合约束特性。

```
CREATE TABLE Persons
(
    P_Id int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255),
    CONSTRAINT pk_PersonID PRIMARY KEY (P_Id,LastName)
)
```

在上面的实例中，只有一个主键 PRIMARY KEY (pk_PersonID)，然而，pk_PersonID 的值是由两个列P_Id 和 LastName) 组成的。

4.4.4 外键约束 foreign key

一个表中的 FOREIGN KEY 指向另一个表中的 PRIMARY KEY.

“Persons” 表中的 “P_Id” 列是 “Persons” 表中的 PRIMARY KEY。

“Orders” 表中的 “P_Id” 列是 “Orders” 表中的 FOREIGN KEY。

FOREIGN KEY 约束用于预防破坏表之间连接的行为。

FOREIGN KEY 约束也能防止非法数据插入外键列，因为它必须是它指向的那个表中的值之一。

```
CREATE TABLE Orders
(
    O_Id int NOT NULL,
    OrderNo int NOT NULL,
    P_Id int,
    PRIMARY KEY (O_Id),
    FOREIGN KEY (P_Id) REFERENCES Persons(P_Id)
)
```


第五章 索引-重要

5.1 简介

索引用于快速找出在某个列中有一特定值的行，不使用索引，MySQL 必须从第一条记录开始读完整个表，直到找出相关的行，表越大，查询数据所花费的时间就越多，如果表中查询的列有一个索引，MySQL 能够快速到达一个位置去搜索数据文件，而不必查看所有数据，那么将会节省很大一部分时间。

例如：有一张 *person* 表，其中有 2W 条记录，记录着 2W 个人的信息。有一个 *Phone* 的字段记录每个人的电话号码，现在想要查询出电话号码为 *xxxx* 的人的信息。

- 如果没有索引，那么将从表中第一条记录一条条往下遍历，直到找到该条信息为止。
- 如果有了索引，那么会将该 **Phone** 字段，通过一定的方法进行存储，好让查询该字段上的信息时，能够快速找到对应的数据，而不必在遍历 2W 条数据了。其中 MySQL 中的索引的存储类型有两种：BTREE、HASH。

5.2 优点、缺点和使用原则

优点

- 所有的 MySQL 列类型 (字段类型) 都可以被索引，也就是可以给任意字段设置索引
- 大大加快数据的查询速度

缺点

- 创建索引和维护索引要耗费时间，并且随着数据量的增加所耗费的时间也会增加
- 索引也需要占空间
- 当对表中的数据进行增加、删除、修改时，索引也需要动态的维护，降低了数据的维护速度。

使用原则 通过上面说的优点和缺点，我们应该可以知道，并不是每个字段度设置索引就好，也不是索引越多越好，而是需要自己合理的使用。

- 对经常更新的表就避免对其进行过多的索引，对经常用于查询的字段应该创建索引
- 数据量小的表最好不要使用索引，因为由于数据较少，可能查询全部数据花费的时间比遍历索引的时间还要短，索引就可能不会产生优化效果
- 在一同值少的列上 (字段上) 不要建立索引，比如在学生表的”性别”字段上只有男，女两个不同值。相反的，在一个字段上不同值较多可是建立索引

5.3 索引的分类

存储引擎支持类型 索引是在存储引擎中实现的，也就是说不同的存储引擎，会使用不同的索引

- MyISAM和InnoDB 存储引擎：只支持 **BTREE** 索引
- MEMORY/HEAP 存储引擎：支持 **HASH** 和 **BTREE** 索引

单列索引 一个索引只包含单个列，但一个表中可以有多个单列索引。主要包括以下几种类型：

1. 普通索引：基本索引类型，没有什么限制，允许在定义索引的列中插入重复值和空值
2. 唯一索引：索引列中的值必须是唯一的，但是允许为空值
3. 主键索引：是一种特殊的唯一索引，不允许有空值。

组合索引 在表中的多个字段组合上创建的索引，只有在查询条件中使用了这些字段的左边字段时，索引才会被使用，使用组合索引时遵循最左前缀集合。

联合索引左侧字段用了范围查询，则其他字段无法用上。

5.4 创建表添加索引

创建普通索引

```
// 方式一
CREATE TABLE book (
  bookid INT NOT NULL,
  bookname VARCHAR(255) NOT NULL,
  authors VARCHAR(255) NOT NULL,
  info VARCHAR(255) NULL,
  comment VARCHAR(255) NULL,
  year_publication YEAR NOT NULL,
  INDEX(year_publication)
```



```
);
```

```
// 方式二
```

```
CREATE TABLE book (  
  bookid INT NOT NULL,  
  bookname VARCHAR(255) NOT NULL,  
  authors VARCHAR(255) NOT NULL,  
  info VARCHAR(255) NULL,  
  comment VARCHAR(255) NULL,  
  year_publication YEAR NOT NULL,  
  KEY(year_publication)  
);
```

创建唯一索引

```
CREATE TABLE t1  
(  
  id INT NOT NULL,  
  name CHAR(30) NOT NULL,  
  UNIQUE INDEX UniqIdx(id)  
);
```

创建主键索引

```
CREATE TABLE t2  
(  
  id INT NOT NULL,  
  name CHAR(10),  
  PRIMARY KEY(id)  
);
```

创建组合索引

```
CREATE TABLE t3  
(  
  id INT NOT NULL,  
  name CHAR(30) NOT NULL,  
  age INT NOT NULL,  
  info VARCHAR(255),  
  INDEX MultiIdx(id,name,age)  
);
```

组合索引就是遵从了最左前缀，利用索引中最左边的列集来匹配行，这样的列集称为最左前缀，例如，这里由 id、name 和 age3 个字段构成的索引，索引行中就按id/name/age 的顺序存放，索引可以索引下面字段组合(id, name, age)、(id, name) 或者(id)。如果要查询的字段不构成索引最左面的前缀，那么就不会是用索引，比如，age 或者 (name, age) 组合就不会使用索引查询

5.5 在已经存在的表上创建索引

```
ALTER TABLE 表名 ADD [UNIQUE|FULLTEXT|SPATIAL] [INDEX|KEY] [索引名] (字段名) [ASC|DESC]  
CREATE [UNIQUE|FULLTEXT|SPATIAL] [INDEX|KEY] 索引名称 ON 表名 (字段名 [length]) [ASC|DESC]
```

- ALTER TABLE book ADD INDEX BkNameIdx(bookname(30));
- CREATE INDEX BkBookNameIdx ON book(bookname);

5.6 删除索引

- ALTER TABLE 表名 DROP INDEX 索引名;
- DROP INDEX 索引名 ON 表名;

5.7 索引背后的算法原理

<http://blog.codinglabs.org/articles/theory-of-mysql-index.html>

5.8 索引设计概要

<https://draveness.me/sql-index-intro>

5.9 参考

<https://www.cnblogs.com/whgk/p/6179612.html>

第六章 事务

第七章 进阶操作

7.1 binlog 日志-重要

<http://www.cnblogs.com/martinzhong/p/3454358.html>

7.1.1 GTID

7.2 备份-重要

- mysqldump: 逻辑备份; 单线程导入导出; 注意设定字符集;
- mydumper: 逻辑备份; “多线程”导入导出; 无需关心字符集;
- innobackup/mysqlbackup: 物理备份; 速度较快;

7.2.1 普通数据库备份

```
mysqldump -u root -p password --default-character-set=utf-8 dataname > dataname.sql  
mysql -u root -p password --default-character-set=utf-8 dataname < dataname.sql
```

7.2.2 备份到压缩文件

```
mysqldump -u root -p database | gzip > database.sql.gz  
gzip < database.sql.gz | mysql -u root -p database
```

7.2.3 增量备份

```
mysqlbinlog bin-log.000002 |mysql -uroot -ppassword
```

7.3 InnoDB 存储引擎

7.4 视图

7.5 主从模式-replication

7.6 集群

7.7 explain 详解

<https://www.cnblogs.com/xuanzhi201111/p/4175635.html>

在日常工作中，我们会有时会开慢查询去记录一些执行时间比较久的 SQL 语句，找出这些 SQL 语句并不意味着完事了，些时我们常常用到 explain 这个命令来查看一个这些 SQL 语句的执行计划，查看该 SQL 语句有没有使用上了索引，有没有做全表扫描，这都可以通过 explain 命令来查看。

explain 出来的信息有 10 列，分别是 id、select_type、table、type、possible_keys、key、key_len、ref、rows、Extra，下面对这些字段出现的可能进行解释：

7.7.1 id

SQL 执行的顺序的标识,SQL 从大到小的执行

- id 相同时，执行顺序由上至下
- 如果是子查询，id 的序号会递增，id 值越大优先级越高，越先被执行
- id 如果相同，可以认为是一组，从上往下顺序执行；在所有组中，id 值越大，优先级越高，越先执行

7.7.2 select_type

- SIMPLE : 简单 SELECT, 不使用 UNION 或子查询等
- PRIMARY : 查询中若包含任何复杂的子部分, 最外层的 select 被标记为 PRIMARY
- UNION : UNION 中的第二个或后面的 SELECT 语句
- DEPENDENT UNION : UNION 中的第二个或后面的 SELECT 语句，取决于外面的查询
- UNION RESULT : UNION 的结果
- SUBQUERY : 子查询中的第一个 SELECT
- DEPENDENT SUBQUERY : 子查询中的第一个 SELECT，取决于外面的查询

- DERIVED : 派生表的 SELECT, FROM 子句的子查询
- UNCACHEABLE SUBQUERY : 一个子查询的结果不能被缓存, 必须重新评估外链接的第一行

7.7.3 table

显示这一行的数据是关于哪张表的

7.7.4 type

- ALL: Full Table Scan, MySQL 将**遍历全表**以找到匹配的行
- Index: Full Index Scan, index 与 ALL 区别为 index 类型**只遍历索引树**
- Range: 只检索给定范围的行, 使用一个索引来选择行
- Ref: 表示上述表的连接匹配条件, 即哪些列或常量**被用于查找索引列上的值**
- Eq_ref: 类似 ref, 区别就在使用的索引是唯一索引, 对于每个索引键值, 表中只有一条记录匹配, 简单来说, 就是多表连接中使用 primary key 或者 unique key 作为关联条件
- Const、System: 当 MySQL 对查询某部分进行优化, 并转换为一个常量时, 使用这些类型访问。如将主键置于 where 列表中, MySQL 就能将该查询转换为一个常量,system 是 const 类型的特例, 当查询的表只有一行的情况下, 使用 system
- NULL: MySQL 在优化过程中分解语句, 执行时甚至不用访问表或索引, 例如从一个索引列里选取最小值可以通过单独索引查找完成。

7.7.5 possible_keys

指出 MySQL 能使用哪个索引在表中找到记录, 查询涉及到的字段上若存在索引, 则该索引将被列出, 但不一定被查询使用

7.7.6 Key

key 列显示 MySQL 实际决定使用的键 (索引)

7.7.7 key_len

表示索引中使用的字节数, 可通过该列计算查询中使用的索引的长度 (key_len 显示的值为索引字段的最大可能长度, 并非实际使用长度, 即key_len 是根据表定义计算而得, 不是通过表内检索出的)

不损失精确性的情况下, 长度越短越好

7.7.8 ref

表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值

7.7.9 rows

表示 MySQL 根据表统计信息及索引选用情况，估算的找到所需的记录所需要读取的行数

7.7.10 总结

- EXPLAIN 不会告诉你关于触发器、存储过程的信息或用户自定义函数对查询的影响情况
- EXPLAIN 不考虑各种 Cache
- EXPLAIN 不能显示 MySQL 在执行查询时所作的优化工作
- 部分统计信息是估算的，并非精确值
- EXPLAIN 只能解释 SELECT 操作，其他操作要重写为 SELECT 后查看执行计划。

7.8 慢查询日志

MySQL 慢查询日志是指 MySQL 中执行时间超过 `long_query_time` 阈值的 SQL 语句。

Dumlo 默认每天零晨 **rotate** 慢查询日志，按日期命名文件，每天一个文件，服务器上保留最近 8 天的慢查询日志文件。

从慢查询日志中统计出执行比较频繁且执行时间比较久、扫描的行数也比较大的 SQL，可以针对这类 SQL 进行优化。

参数说明 慢查询日志相关的几个参数，有先后依赖关系，顺序如下：

1. `slow_query_log`

是否启用慢查询日志功能，慢查询日志的内容可以在 **Dumlo 管理平台** 实例管理 -> 日志管理 -> 慢查询日志上查看

2. `log_slow_admin_statements`

是否记录 administrative statements 语句到慢查询日志里。

3. `long_query_time`

定义慢查询日志的触发条件，如果 SQL 实际执行时间（不包括锁的时间）超过 `long_query_time` 定义的阈值，此 SQL 则被记录到慢查询日志里。

4. `min_examined_row_limit`

只有慢查询语句的执行行数检查返回大于该参数指定值，此慢查询语句才被记录到慢查询日志中。

表 7.1: 慢查询日志参数

方法	MySQL 默认值	Dumbo 默认值	能否自助修改
<code>slow_query_log</code>	OFF	ON	否
<code>long_query_time</code>	10s	0.1s	能
<code>log_slow_admin_statements</code>	OFF	ON	否
<code>log_slow_slave_statements</code>	OFF	ON	否
<code>min_examined_row_limit</code>	0	100	否

因此 Dumbo 实例默认配置下，记录慢查询日志的条件是：SQL 实际执行时间超过 0.1s，且此 SQL 的执行行数检查返回值大于 100。

优点

- 语句调优
通过统计慢查询日志（可联系 Dumbo 值班），找出最耗时的 top 10 语句，优化完成后，再统计新的慢查询日志，一直重复，直到没有产生新的慢查询语句。
- 性能排查
如果一个实例的 CPU 使用率比较高，可以优先检查一下该实例的慢查询日志，如果慢查询日志刷得比较厉害，可以初步确认是由于 SQL 语句不够优化导致实例 CPU 使用率高，可以通过优化 SQL 来解决。

7.9 查询优化-海量数据

7.10 经验 Tips

<https://coolshell.cn/articles/1846.html>

1. **EXPLAIN SELECT** 查询
解释查询语句，查看是否使用到索引 (**Key**)
2. 当只要一行数据时使用 **LIMIT 1**
这样会在找到一条数据后停止搜索，而不是继续往后查下一条符合记录的数据
3. 为搜索字段建索引
BTree Or Hash Better Than All
4. 在 Join 表的时候使用相当类型的例，并将其索引
5. 千万不要 **ORDER BY RAND()**
6. 避免 **SELECT ***

7. 永远为每张表设置一个 ID
8. 使用 `ENUM` 而不是 `VARCHAR`
9. 尽可能的使用 `NOT NULL`
10. 无缓冲的查询
11. 把 IP 地址存成 `UNSIGNED INT`
12. 固定长度的表会更快
13. 垂直分割
14. 拆分大的 `DELETE` 或 `INSERT` 语句
15. 越小的列会越快
16. 选择正确的存储引擎

第八章 MySQL 性能优化



图 8.1: 性能优化结构

8.1 影响性能的因素

8.1.1 不合理的需求如何优化

需求：一个论坛帖子总量的统计附加要求：实时更新

- 初始阶段：select count(id)
- 新建一个表，在这个表中更新这个汇总数据（频率问题- update 锁）
- 真正的问题在于，实时？创建一个统计表，隔一段时间统计一次并存入 (Redis)。

8.1.2 无用功能的堆积

- 无用的列堆积
- 错误的表设计
- 无用的表关联

8.1.3 哪些数据不适合放在数据库中

- 二进制文件-文件-图片
- 流水队列数据
- 超大文本

8.1.4 合理的 cache

哪些数据适合放到 cache 中

- 系统的配置信息
- 活跃用户的基本信息
- 活跃用户的定制化信息
- 基于时间段的统计信息
- 读 »> 写的数据库

减少数据库交互次数

减少重复执行相同的 SQL

8.1.5 其他

- cache 系统的不合理利用导致 Cache 命中率底下造成的数据库访问量的增加，同事也浪费了 Cache 系统的硬件资源投入
- 过度依赖面向对象思想，对系统的可扩展性的过度追求，促使系统设计的时候将对象拆的过于分散，造成系统中大量的复杂 join 语句，而 MySQL 在各数据库系统中的主要优势在于处理简单逻辑的查询，这与其锁定的机制也有较大关系
- 对数据库过度依赖，将大量适合存放于文件系统中的数据存入了数据库中，造成数据库资源的浪费，影响到系统的整体性能，如各种日志信息
- 过度理想化系统的用户体验，是大量的非核心业务消耗过多的资源，如大量

8.2 Sql 优化

8.3 Mysql 锁

8.4 MySQL 复制

8.5 其他优化