MySQL 数据库学习笔记

郑华

2019 年 4 月 25 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。

系统博客：https://www.cnblogs.com/geaozhang/category/1326927.html
周伯通博客：https://www.cnblogs.com/phpper/tag/mysql/default.html?page=1

1.2　数据类型介绍

https://www.cnblogs.com/-xlp/p/8617760.html#undefined

- **整数类型**：BIT、BOOL、TINY INT、SMALL INT、MEDIUM INT、INT、BIG INT
- **浮点数类型**：FLOAT、DOUBLE、DECIMAL
- **字符串类型**：CHAR、VARCHAR、TINY TEXT、TEXT、MEDIUM TEXT、LONGTEXT、TINY BLOB、BLOB、MEDIUM BLOB、LONG BLOB

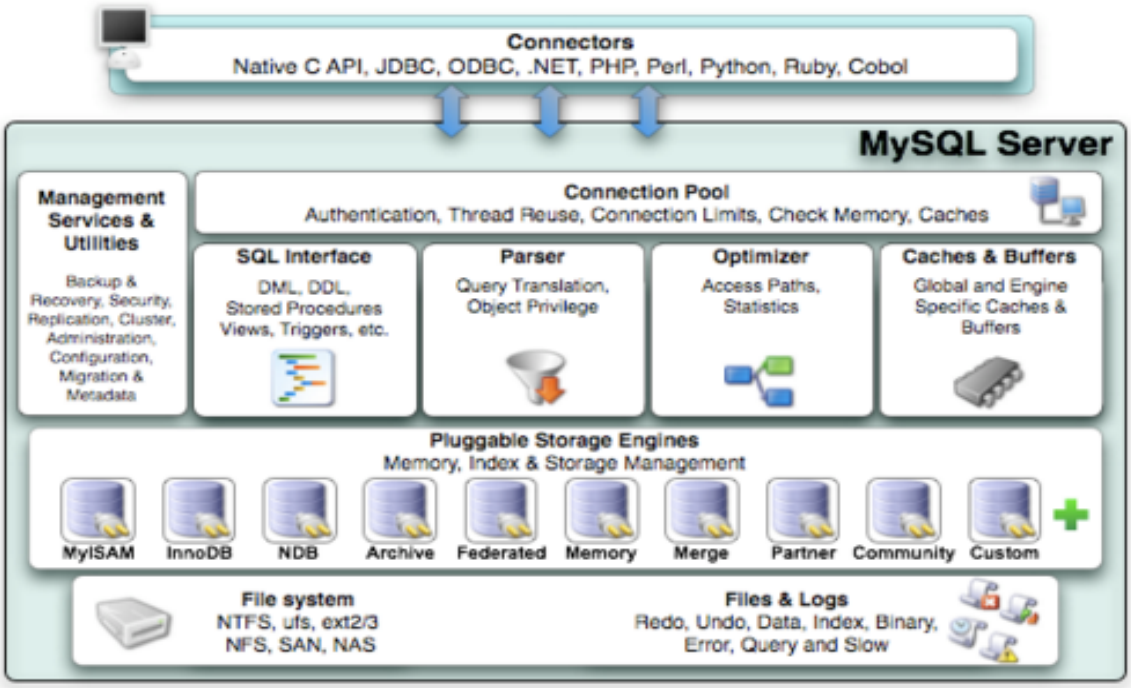
- **日期类型**：Date、DateTime、TimeStamp、Time、Year
- **其他数据类型**：BINARY、VARBINARY、ENUM、SET、Geometry、Point、MultiPoint、LineString、MultiLineString、Polygon、GeometryCollection 等

1.3 Mysql 架构

https://www.cnblogs.com/andy6/p/6626848.html

<MySql 技术内幕 >: https://www.cnblogs.com/lay2017/p/9165203.html

1.3.1 MySQL 层次结构



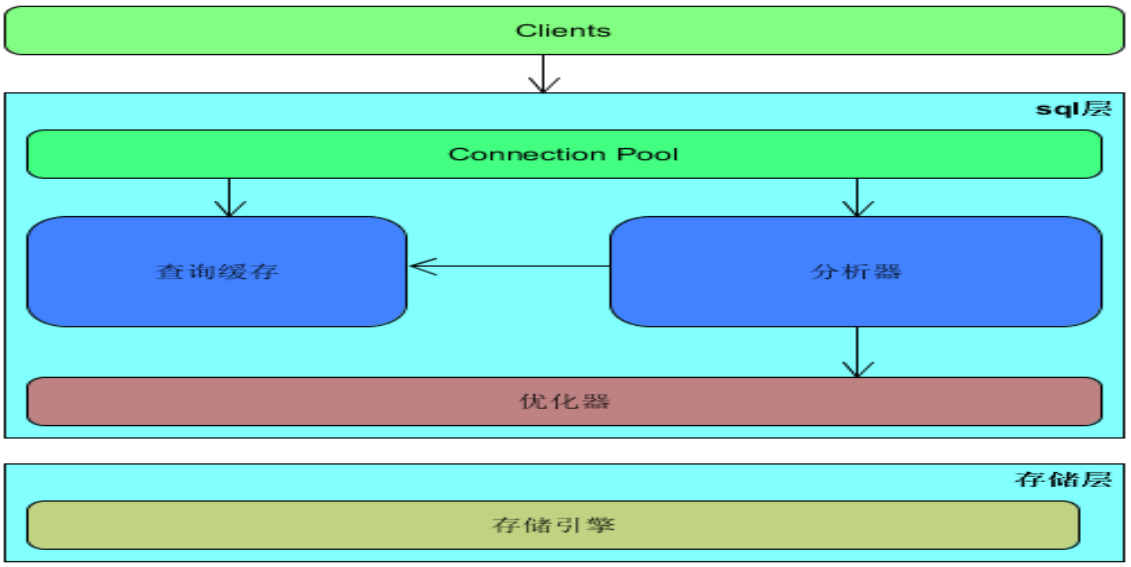


图 1.1: mysql 架构

Connectors 指的是不同语言中与 SQL 的交互。

Management Serveices & Utilities: 系统管理和控制工具。

Connection Pool: 连接池。管理缓冲用户连接，线程处理等需要缓存的需求。

`SQL Interface`：SQL 接口。接受用户的 SQL 命令，并且返回用户需要查询的结果。比如 select from 就是调用 SQL Interface。

`Parser`：解析器。SQL 命令传递到解析器的时候会被解析器验证和解析。解析器是由 Lex 和 YACC 实现的，是一个很长的脚本。主要功能：

- 将 SQL 语句分解成数据结构，并将这个结构传递到后续步骤，以后 SQL 语句的传递和处理就是基于这个结构的。
- 如果在分解构成中遇到错误，那么就说明这个 sql 语句是不合理的。

`Optimizer`: 查询优化器。

`Cache`和`Buffer`：查询缓存。

- 如果查询缓存有命中的查询结果，查询语句就可以直接去查询缓存中取数据。这个缓存机制是由一系列小缓存组成的。比如表缓存，记录缓存，key 缓存，权限缓存等。

`Engine` ： 存储引擎。

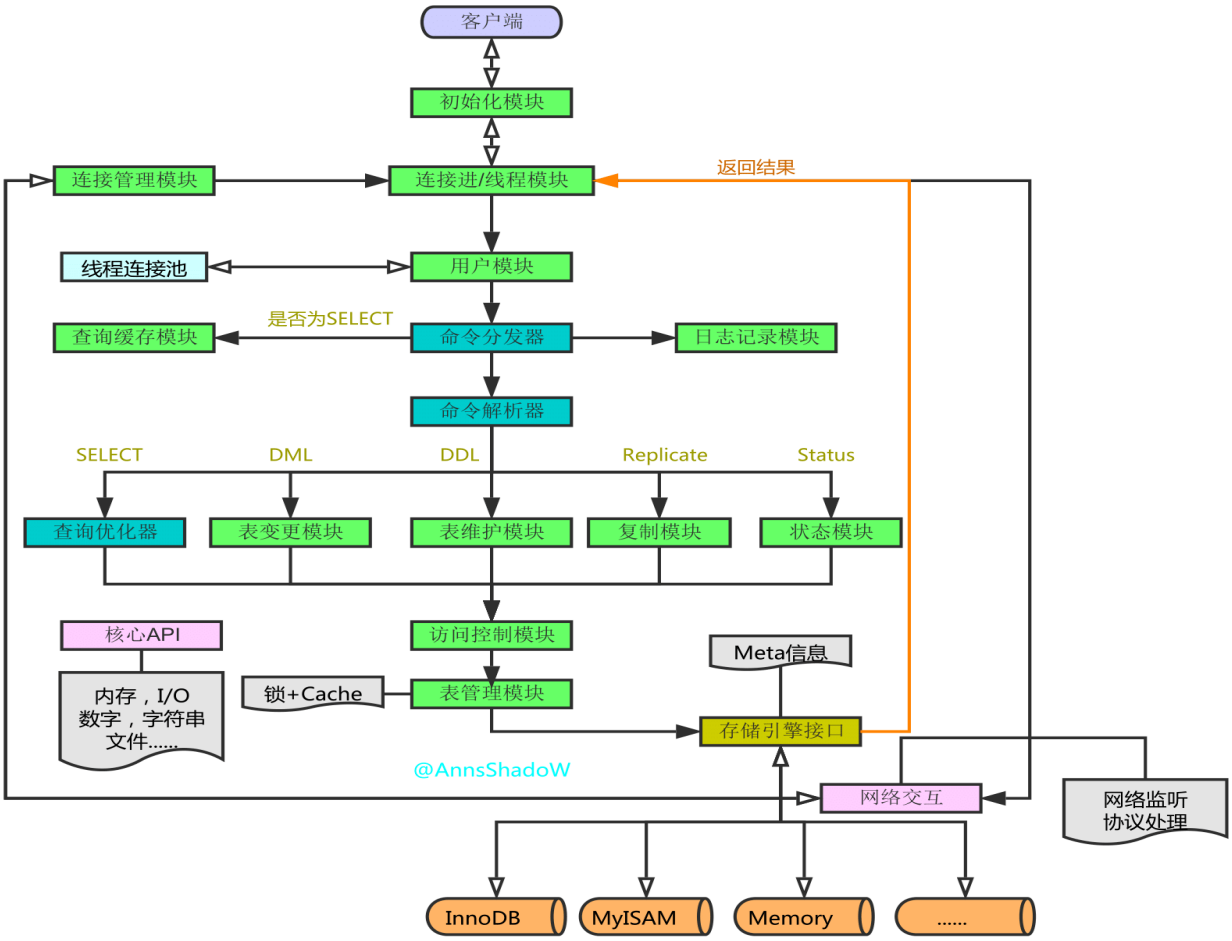


图 1.2: mysql 架构

`mysql` 是一个 **C/S 架构模型**，客户端通过与服务端建立连接来操作服务端数据；
分析器**分析请求**，并转发给优化器；

通过缓存的方式提高查询性能；
优化器负责和底层的存储引擎进行交互，存储和查询 mysql 的数据；

1.3.2 物理文件组成

- 日志文件：
  - Error log 错误日志：记录遇到的所有严重的错误信息、每次启动关闭的详细信息；
  - Binary log 二进制日志：也就是 binlog，记录所有修改数据库的操作；
  - Query log 查询日志：记录所有查询操作，体积较大，开启后对性能有影响；
  - Slow Query log 慢查询日志：记录所有执行时间超过long_query_time 的 sql 语句和达到min_examined_row_limit 条距离的语句；
  - InnoDB redo log：记录 InnoDB 所做的物理变更和事务信息；
- 数据文件：
  - .frm文件：表结构定义信息
  - .MYD文件：MyISAM 引擎的数据文件；
  - .MYI文件：MyISAM 引擎的索引文件；
  - .ibd文件和.ibdata文件：**InnoDB 的数据和索引**；.ibdata 配置为共享表空间时使用，.ibd 配置为独享表空间时使用；
- 其他文件：
  - 系统配置文件：/etc/my.cnf
  - pid文件：存储自己的进程 ID
  - Unix Socket文件：连接客户端使用

1.4 教程集合地址

https://blog.csdn.net/orangleliu/article/details/54694272
DB 入门笔记：https://www.cnblogs.com/ggjucheng/archive/2012/11/02/2751119.html

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

2.3　MySql 查询执行流程

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

2.3.1　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

2.3.2 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

2.3.3 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

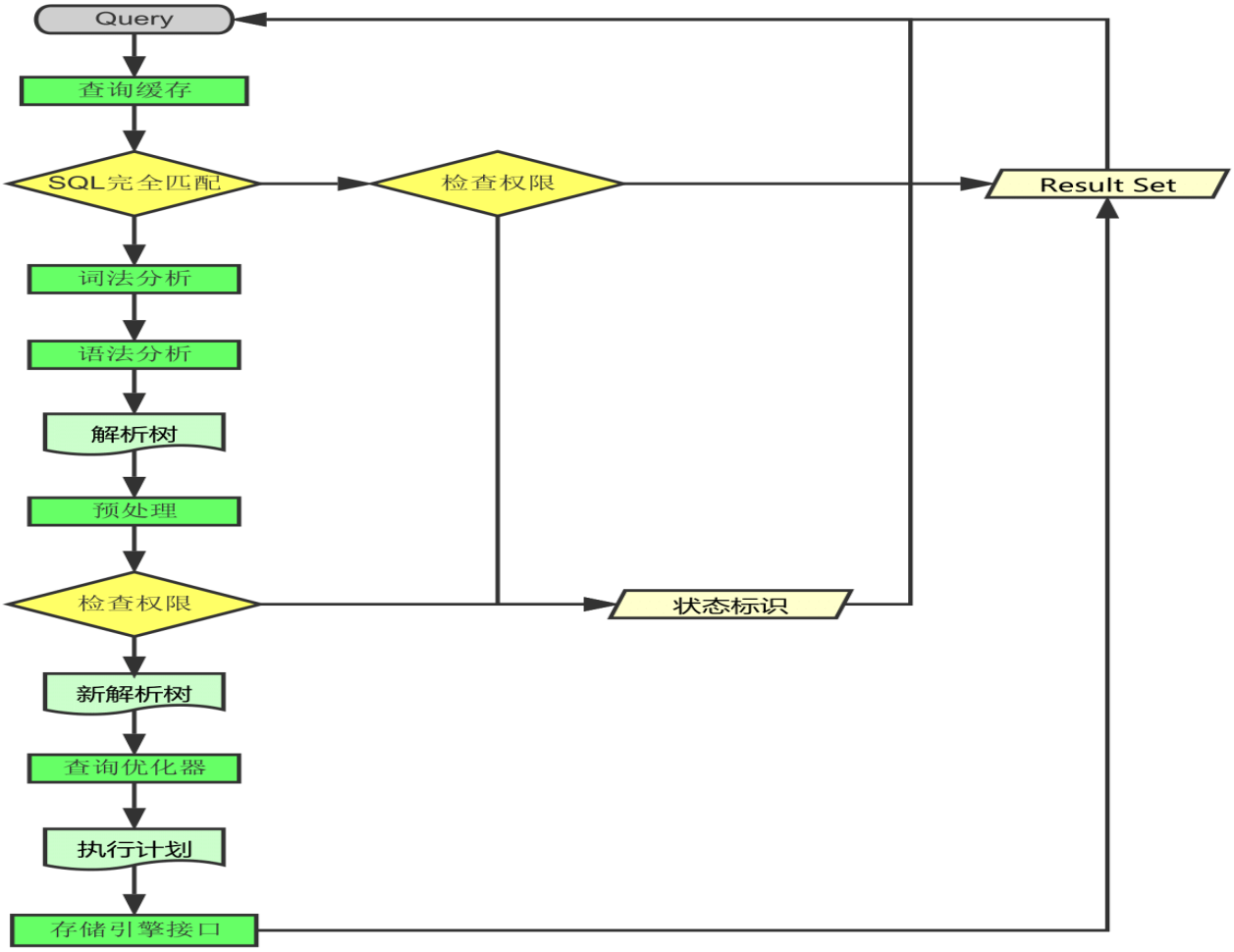

Query
查询缓存
SQL完全匹配
检查权限
Result Set
词法分析
语法分析
解析树
预处理
检查权限
状态标识
新解析树
查询优化器
执行计划
存储引擎接口

图 2.1: 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长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】

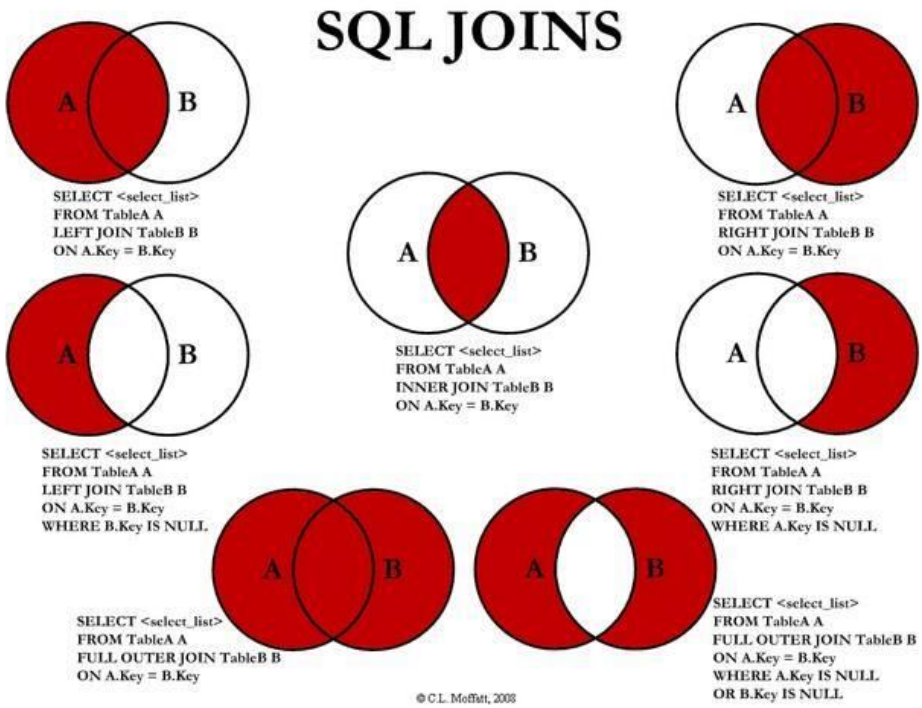


图 3.1: SQL Joins

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。

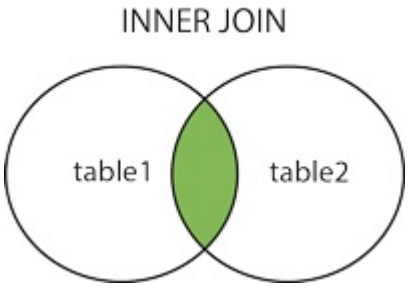


图 3.2: 内连接

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.depno = d.depno
```

```
// 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, 与其的薪水等级
    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 >= 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。

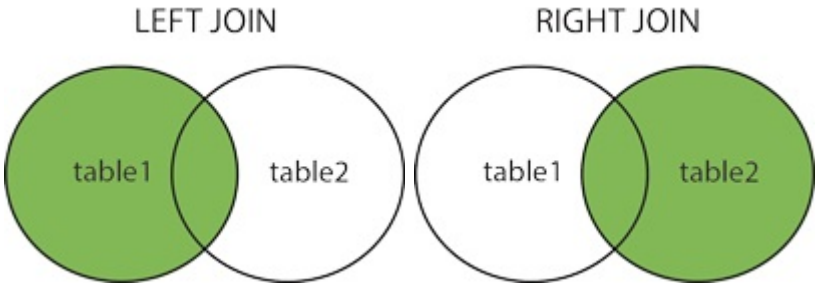


图 3.3: 外连接 (左右)

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

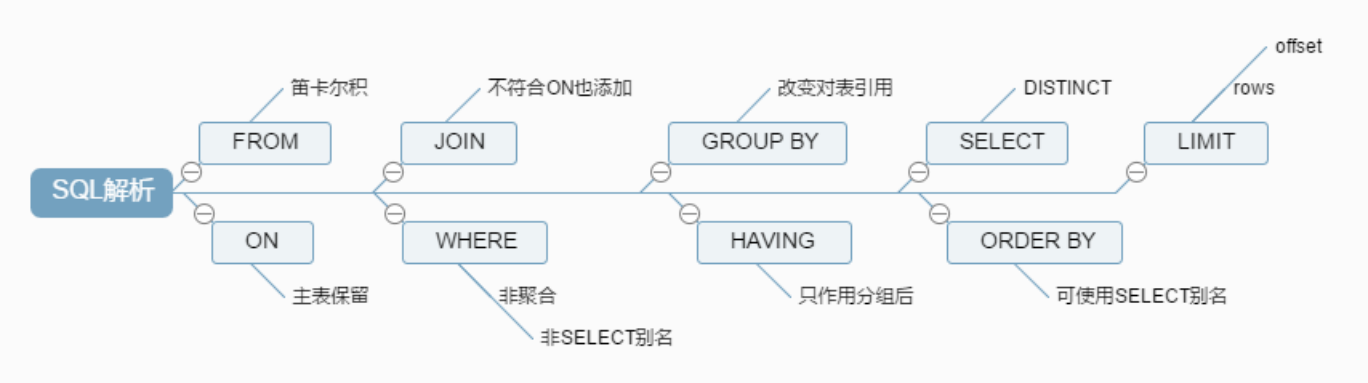


图 3.4: SQL 解析顺序

```
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`

3.9 视图 View

一种虚拟的表，将查询封装到该视图中。

视图可以包含表的全部或者部分记录，也可以由一个表或者多个表来创建。使用视图就可以不用看到数据表中的所有数据，而是只想得到所需的数据。当我们创建一个视图的时候，**实际上是**在数据库里执行了`SELECT` 语句，`SELECT` 语句包含了字段名称、函数、运算符，来给用户显示数据。

视图在外观上和表很相似，但是它不需要实际上的物理存储，**数据还是存储在原来的表里**。在数据库中，只存放了视图的定义。

视图的**使用方式与表的使用方式一致**。

基于视图可以创建视图。

视图**增加了数据的安全性和逻辑独立性**，数据库的设计和结构不会受到视图中的函数、where 或 join 语句的影响。视图可以只展现数据表的一部分数据，对于我们不希望让用户看到全部数据，只希望用户看到部分数据的时候，可以选择使用视图。

更新视图可以更新真实表。

3.9.1 创建

```
CREATE [ALGORITHM = {MERGE  | TEMPTABLE | UNDEFINED}]  VIEW 视图名称[(
    column_list)] AS SELECT 语句  WITH  [CASCADED|LOCAL] CHECK OPTION

create view employee_view as SELECT * from employee;
```

首先，第一个中括号里代表的就是创建视图是的算法属性，它允许我们控制mysql 在创建视图时使用的机制，并且mysql 提供了三种算法：MERGE，TEMPTABLE 和 UNDEFINED。

- **使用 MERGE 算法**，mysql 首先**将输入查询与定义视图的 select 语句组合成单个查询。然后 mysql 执行组合查询返回结果集**。如果 select 语句**包含集合函数** (如 min，max，sum，count，avg 等) **或** distinct，group by，havaing，limit，union，union all，**子查询，则不允许使用 MERGE 算法**。如果 select **语句无引用表，则也不允许使用 MERGE 算法**。如果不允许 MERGE 算法，mysql 将算法更改为 UNDEFINED。我们要注意，将视图定义中的输入查询和查询组合成一个查询称为视图分辨率。

- **使用 TEMPTABLE 算法**，mysql 首先**根据定义视图的 SELECT 语句创建一个临时表，然后针对该临时表执行输入查询**。因为 mysql 必须创建临时表来存储结果集并将数据从基表移动到临时表，所以 TEMPTABLE 算法的效率比 MERGE 算法效率低。另外，**使用 TEMPTABLE 算法的视图是不可更新的**。

- 当我们创建视图而不指定显式算法时，**UNDEFINED 是默认算法**。UNDEFINED 算法使 mysql 可以选择使用 MERGE 或 TEMPTABLE 算法。mysql 优先使用 MERGE 算法，因为 MERGE 算法效率更高。

- CASCADED 默认值，表示更新视图的时候，要满足视图和表的相关条件

- LOCAL：表示更新视图的时候，要满足该视图定义的一个条件即可

with check option：对视图进行更新操作的时，需要检查更新后的值是否还是满足视图公式定义的条件。通俗点，就是所更新的结果是否还会在视图中存在。如果更新后的值不在视图范围内，就不允许更新如果创建视图的时候，没有加上with check option，更新视图中的某项数据的话，mysql 并不会进行有效性检查。删掉了就删掉了。在视图中将看不到了。所以使用 WHIT [CASCADED|LOCAL] CHECK OPTION 选项可以保证数据的安全性

3.9.2 查看视图数据

```
SELECT * FROM employee_view;
```

3.9.3 查看视图

```
show CREATE view employee_view;
```

3.9.4 删除视图

```
drop view employee_view
```

3.9.5 修改视图

```
create or replace view employee_view as select eid,ename,salary FROM
    employee;

alter view employee_view as SELECT * FROM employee;
```

3.9.6 修改视图中的数据

```
UPDATE employee_view set ename='小红' WHERE ename='小个';
```

第四章　增、删、改

4.1　表、(with 约束)

4.1.1　create table tableName(columName 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 newColumeName type(length);

4.2.2 alter table tableName modify column newType(lenght);

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)
)
```

4.5 触发器 Trigger

触发器**是与表有关的数据库对象**，在满足定义条件时触发，并执行触发器中定义的语句集合。触发器的这种特性可以协助应用在数据库端确保数据的完整性。

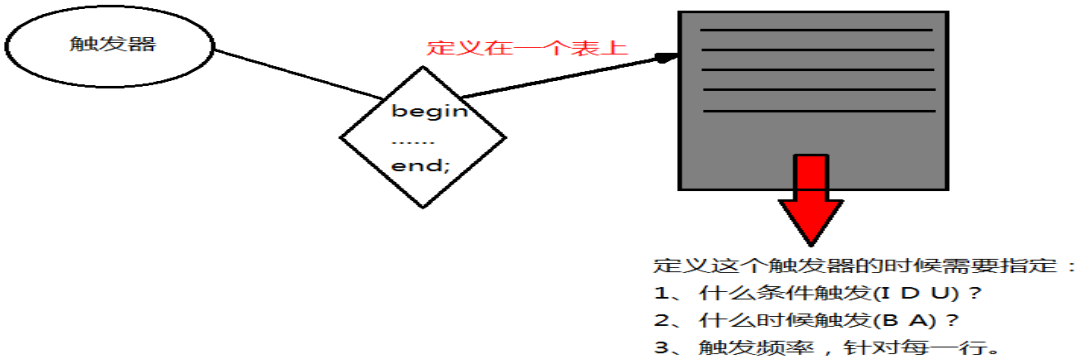


图 4.1: 触发器结构演示

特性

- 有begin end 体，begin end; 之间的语句可以写的简单或者复杂
- **什么条件**会触发：Insert、Update、Delete
- **什么时候**触发：在增删改**前或者**后
- 触发频率：**针对每一行执行**
- 触发器定义在表上，附着在表上。
- **cannot** associate a **Trigger** with a **TEMPORARY table** or a **View**.
- 触发器是针对每一行的；对增删改非常频繁的表上切记不要使用触发器，因为它会非常消耗资源。

4.5.1 创建触发器

```
CREATE
    [DEFINER = { user | CURRENT_USER }]
TRIGGER trigger_name
trigger_time trigger_event
ON tbl_name FOR EACH ROW
    [trigger_order]
trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }

trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```

FOR EACH ROW 表示任何一条记录上的操作满足触发事件都会触发该触发器，也就是说触发器的**触发频率是针对每一行数据触发一次**。

创建只有一个执行语句的触发器

```
CREATE TRIGGER 触发器名 BEFORE|AFTER 触发事件 ON 表名 FOR EACH ROW 执行语句;
```

创建有多个执行语句的触发器

```
CREATE TRIGGER 触发器名 BEFORE|AFTER 触发事件
ON 表名 FOR EACH ROW
BEGIN
        执行语句列表
END;
```

4.5.2 查看触发器

- SHOW TRIGGERS;
- SELECT * FROM information_schema.triggers;

4.5.3 删除触发器

```
DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name
```

第五章　存储过程

http://www.runoob.com/w3cnote/mysql-stored-procedure.html

存储过程就**类似于脚本**。

存储过程是**为了完成特定功能的 SQL 语句集**，经编译创建并保存在数据库中，用户可通过指定存储过程的名字并给定参数 (需要时) 来调用执行。

5.1　存储过程的创建和调用

```
CREATE
    [DEFINER = { user | CURRENT_USER }]
  PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body

proc_parameter:
    [ IN | OUT | INOUT ] param_name type

characteristic:
    COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }

routine_body:
    Valid SQL routine statement

[begin_label:] BEGIN
    [statement_list]
        ……
END [end_label]
```

5.1.1　声明语句结束符

```
DELIMITER $$
或
DELIMITER //
```

5.1.2 声明存储过程

```
CREATE PROCEDURE demo_in_parameter(IN p_in int)
```

5.1.3 存储过程开始和结束符号

```
BEGIN .... END
```

5.1.4 变量赋值

```
SET @p_in=1
```

5.1.5 变量定义

```
DECLARE l_int int unsigned default 4000000;
```

5.1.6 创建存储过程、存储函数名 (参数)

```
create procedure 存储过程名(参数)
```

5.1.7 存储过程体

```
create function 存储函数名(参数)
```

5.1.8 实例

```
mysql> delimiter $$    #将语句的结束符号从分号;临时改为两个$$(可以是自定义)
mysql> CREATE PROCEDURE delete_matches(IN p_playerno INTEGER)
    -> BEGIN
    ->     DELETE FROM MATCHES
    ->    WHERE playerno = p_playerno;
    -> END$$
Query OK, 0 rows affected (0.01 sec)

mysql> delimiter;    #将语句的结束符号恢复为分号
```

```
mysql> select * from MATCHES;
+---------+--------+----------+-----+------+
| MATCHNO | TEAMNO | PLAYERNO | WON | LOST |
+---------+--------+----------+-----+------+
|       1 |      1 |        6 |   3 |    1 |
|       7 |      1 |       57 |   3 |    0 |
|       8 |      1 |        8 |   0 |    3 |
```

```
|        9 |      2 |        27 |   3 |    2 |
|       11 |      2 |       112 |   2 |    3 |
+----------+--------+-----------+-----+------+
5 rows in set (0.00 sec)

mysql> call delete_matches(57);
Query OK, 1 row affected (0.03 sec)

mysql> select * from MATCHES;
+---------+--------+----------+-----+------+
| MATCHNO | TEAMNO | PLAYERNO | WON | LOST |
+---------+--------+----------+-----+------+
|       1 |      1 |        6 |   3 |    1 |
|       8 |      1 |        8 |   0 |    3 |
|       9 |      2 |       27 |   3 |    2 |
|      11 |      2 |      112 |   2 |    3 |
+---------+--------+----------+-----+------+
4 rows in set (0.00 sec)
```

5.2 存储过程的参数

5.3 存储过程的变量

第六章　游标

有数据缓冲的思想：游标的设计**是一种数据缓冲区的思想，用来存放 SQL 语句执行的结果**。

游标是在先从数据表中检索出数据之后才能继续灵活操作的技术。类似于指针：游标类似于指向数据结构堆栈中的指针，用来 pop 出所指向的数据，并且只能每次取一个。

6.1　特点

- 游标是**针对行操作的**，所以对从数据库中 select 查询得到的每一行可以进行分开的独立的相同或不同的操作，是一种分离的思想。
- 在数据量大的情况下，是不适用的，速度过慢。这里有个比喻就是：当你去 ATM 存钱是希望一次性存完呢，还是 100 一张一张的存，这里的 100 一张一张存就是游标针对行的操作。数据库大部分是面对集合的，业务会比较复杂，而游标使用**会有死锁**，影响其他的业务操作，不可取。当数据量大时，使用游标会造成**内存不足**现象。

6.2　创建与使用

6.2.1　定义游标

```
DECLARE <游标名> CURSOR FOR select语句;

DECLARE mycursor CURSOR FOR select * from shops_info;
```

6.2.2　打开游标

```
open <游标名>
```

6.2.3　使用游标

使用游标需要用关键字 fetch 来取出数据，然后取出的数据需要有存放的地方，我们需要用 declare 声明变量存放列的数据其语法格式为：

```
declare 变量1 数据类型(与列值的数据类型相同)
declare 变量2 数据类型(与列值的数据类型相同)
declare 变量3 数据类型(与列值的数据类型相同)
```

```
FETCH [NEXT | PRIOR | FIRST | LAST] FROM <游标名> [ INTO 变量名1,变量名2,变量
    名3[,…] ]
```

```
-- 声明四个变量
  declare id varchar(20);
  declare pname varchar(20);
  declare pprice varchar(20);
  declare pdescription varchar(20);

-- 1、定义一个游标mycursor
  declare mycursor cursor for
    select *from shops_info;
-- 2、打开游标
  open mycursor;
-- 3、使用游标获取列数据放入变量中
  fetch  next from mycursor into id,pname,pprice,pdescription;
```

6.2.4 关闭游标

```
close mycursor;
```

6.2.5 释放游标

```
deallocate mycursor;
```

6.3 实例

https://www.cnblogs.com/mqxs/p/6018766.html

https://www.cnblogs.com/progor/p/8875100.html

6.3.1 普通游标

```
drop procedure if exists cursor_test;
delimiter //
create procedure cursor_test()
begin
    -- 声明与列的类型相同的四个变量
    declare id varchar(20);
    declare pname varchar(20);
    declare pprice varchar(20);
    declare pdescription varchar(20);

-- 1、定义一个游标mycursor
    declare mycursor cursor for
```

```
    select *from shops_info;
-- 2、打开游标
    open mycursor;
-- 3、使用游标获取列的值
    fetch  next from mycursor into id,pname,pprice,pdescription;
-- 4、显示结果
    select id,pname,pprice,pdescription;
-- 5、关闭游标
    close mycursor;
end;
//
delimiter ;
call cursor_test();
```

当然可以使用循环，while 循环定义如下：

```
WHILE expression DO
    Statements;
END WHILE
//实例
DECLARE num INT;
DECLARE my_string  VARCHAR(255);
SET num =1;
SET str ='';
  WHILE num  < span>10DO
SET  my_string =CONCAT(my_string,num,',');
SET  num = num +1;
END WHILE;
```

6.3.2 循环游标

```
create procedure p3()
begin
    declare id int;
    declare name varchar(15);
    declare flag int default 0;
    -- 声明游标
    declare mc cursor for select * from class;
    declare continue handler for not found set flag = 1;
    -- 打开游标
    open mc;
    -- 获取结果
    l2:loop

    fetch mc into id,name;
    if flag=1 then -- 当无法fetch会触发handler continue
        leave l2;
    end if;
```

```
    -- 这里是为了显示获取结果
    insert into class2 values(id,name);
    -- 关闭游标
    end loop;
    close mc;

end;

call p3();-- 不报错
select * from class2;
```

第七章　索引-重要

7.1　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。

7.2　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

7.3 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。

7.4 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**

7.5 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);

7.6 删除索引

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

7.7 索引背后的算法原理

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

7.8 索引设计概要

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

7.9 参考

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

第八章　事务

8.1　概述

一般来说，事务是必须满足 4 个条件（ACID）:：原子性（Atomicity，或称不可分割性）、一致性（Consistency）、隔离性（Isolation，又称独立性）、持久性（Durability）。

- **原子性**：一个事务（*transaction*）中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
- **一致性**：在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的预设规则，这包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。
- **隔离性**：数据库允许多个并发事务同时对其数据进行读写和修改的能力，隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致。事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。
- **持久性**：事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。

8.2　应用

用 BEGIN, ROLLBACK, COMMIT 来实现.
BEGIN 开始一个事务
ROLLBACK 事务回滚
COMMIT 事务确认

```
mysql> begin;  # 开始事务
Query OK, 0 rows affected (0.00 sec)

mysql> insert into runoob_transaction_test value(5);
Query OK, 1 rows affected (0.01 sec)

mysql> insert into runoob_transaction_test value(6);
Query OK, 1 rows affected (0.00 sec)
```

```
mysql> commit; # 提交事务
Query OK, 0 rows affected (0.01 sec)

mysql>  select * from runoob_transaction_test;
+------+
| id   |
+------+
| 5    |
| 6    |
+------+
2 rows in set (0.01 sec)

mysql> begin;    # 开始事务
Query OK, 0 rows affected (0.00 sec)

mysql>  insert into runoob_transaction_test values(7);
Query OK, 1 rows affected (0.00 sec)

mysql> rollback;   # 回滚
Query OK, 0 rows affected (0.00 sec)

mysql>   select * from runoob_transaction_test;   # 因为回滚所以数据没有插入
+------+
| id   |
+------+
| 5    |
| 6    |
+------+
2 rows in set (0.01 sec)
```

第九章　日志

9.1　bin-log

MySQL 的二进制日志可以说是 MySQL **最重要的日志**了,它记录了所有的 **DDL 和 DML**(除了数据查询语句) 语句，**以事件形式记录，还包含语句所执行的消耗的时间**，MySQL 的二进制日志是事务安全型的。

二进制有两个最重要的使用场景:

- 其一：MySQL Replication **在 Master 端开启 binlog**，*Master 把它的二进制日志传递给 slaves* 来达到 master-slave 数据一致的目的。
- 其二：自然就是数据恢复了，通过使用 mysqlbinlog 工具来使恢复数据。

参考　https://www.cnblogs.com/martinzhang/p/3454358.html

9.1.1　GTID

9.2　error-log

9.3　slow-query-log

MySQL **慢查询日志**是指 MySQL 中**执行时间超过** `long_query_time` **阀值的 SQL 语句**。

Dumbo 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 **Dumbo 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` 定义的阀值，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 9.1: 慢查询日志参数

方法	MySQL 默认值	Dumbo 默认值	能否自助修改
`slow_query_log`	OFF	ON	否
`long_query_time`	10s	0.1s	能
`log_slow_admin_statements`	OFF	ON	否
`log_slow_slave_statements`	OFF	ON	否
`min_examined_row_limit`	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。

参考

- https://www.cnblogs.com/saneri/p/6656161.html

第十章　存储引擎

10.1　InnoDB 存储引擎

第十一章　集群

11.1　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;

11.1.1　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
```

11.1.2　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
```

11.1.3　增量备份

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

11.2　主从模式-replication

11.3　集群

第十二章　MySQL 性能优化

图 12.1: 性能优化结构

12.1　影响性能的因素

12.1.1　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)。

12.1.2　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

12.1.3 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

12.1.4 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

减少数据库交互次数

减少重复执行相同的 SQL

12.1.5 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

12.2 Sql 优化

12.3 Mysql 锁

12.4 MySQL 复制

12.5 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。

expain 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：

12.5.1 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

12.5.2 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

12.5.3 table

显示这一行的数据是关于哪张表的

12.5.4 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。

12.5.5 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

12.5.6 Key

key 列显示 MySQL 实际决定使用的键（索引）

12.5.7 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

12.5.8 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

12.5.9 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

12.5.10 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
- EXPALIN 只能解释 SELECT 操作，其他操作要重写为 SELECT 后查看执行计划。

12.6 查询优化-海量数据

12.7 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，而不是继续往后查少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

第十三章　疑难杂症

13.1　mysql.sock -ERROR 2002 (HY000)

Can't connect to local MySQL server through socket '/tmp/mysql.sock' (2)

https://blog.csdn.net/hjf161105/article/details/78850658