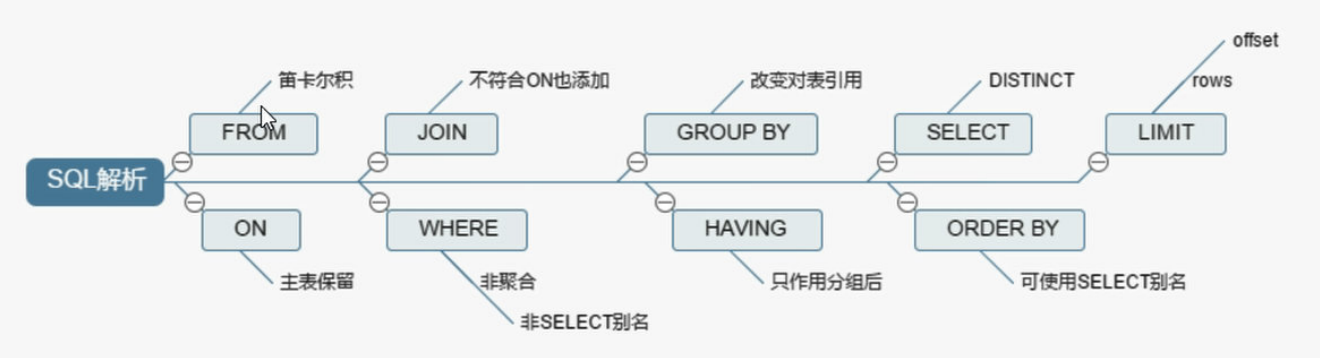
MySQL性能

一、MYSQL的SQL性能下降原因

- 查询语句写的很烂（尽量不使用全列）
  - in、not in
  - >、<
- 索引失效(单值索引和复合索引)
  - 假如说一个用户表 id,name,email,qqcode。
  - 单值索引--------create index idx_user_name on user(name).
  - 复合索引--------create index idx_user_nameemail on user(user,email)
- 关联查询太多join
  - SQL解析执行的过程如下：



  - 关联查询 left join on ,right join on inner join on / full join on 七种形态。
- 网卡流量
  - 减少从服务器的数量
  - 进行分级缓存
  - 避免使用“select *”进行查询
  - 分离业务网络和服务器网络
- 磁盘I/O
  - 磁盘IO性能突然下降（使用更快的磁盘设备）
  - 其它大量消耗磁盘性能的计划任务（调整计划任务，做好磁盘维护）
- 服务器硬件
- SQL查询速度
- 大量的并发和超高的CPU使用率
  - 大量的并发：数据库连接数被占满（max_connections默认100）
  - 超高CPU使用率：因CPU资源消耗而出现宕机
- 大表
  - 概念
    - 记录行数巨大，单表超过千万行

        - 表数据文件巨大，表数据文件超过10G
    - 影响
        - 慢查询：很难在一定的时间内过滤出所需要的数据
        - 对DDL操作的影响
            - 如建立索引需要很长的时间
                - MySQL版本<5.5建立索引会锁表
                - MySQL版本>=5.5虽然不会锁表但会引起主从延迟
            - 修改表结构需要长时间锁表
                - 会造成长时间的主从延迟
                - 影响正常的数据操作
    - 如何处理数据库中的大表
        - 分库分表：把一张大表分成多个小表
            - 难点
                - 分表主键的选择
                - 分表后跨分区数据的查询和统计
        - 历史归档：减少对前后端业务的影响
            - 难点
                - 归档时间点的选择
                - 如何进行归档操作
- 大事务
    - 概念
        - 运行时间比较长，操作的数据比较多的事务
    - 特性
        - 原子性
        - 一致性
        - 隔离性
        - 持久性
    - 风险
        - 锁定太多的数据，造成大量的阻塞和锁超时
        - 回滚时所需时间比较长
        - 执行时间长，容易造成主从延迟
    - 如何处理大事务
        - 避免一次处理太多的数据
        - 移出不必要在事务中的SELECT操作

二、索引

1、概念

- MYSQL官方对索引的定义为：**索引（Index）是帮助MYSQL高效获取数据的数据结构**。从而可以得到索引的本质：索引是数据结构。（索引文件）
- 索引的目的在于提高查询效率，可以类比字典。
- 比如：如果要查找“mysql”。我们肯定需要定位到m字母，然后在往下找到y字母，在找剩下的sql。
- 如果没有索引，那么你可能需要a---z，逐条查询，如果我们想找到java或者oracle开头的词条呢?
- 是不是觉得如果没有索引，这个事情根本没办法完成呢?
- 你可以理解为：排好序的快速查找数据结构。

1）数据本身之外，数据库还维护中一个满足特点查找算法的数据结构，这些数据结构以某种方式指向数据，这样就可以在这些数据结构的基础上实现高级查找算法，这种数据结构就是索引。

2）索引本身就很大，不可能全部存储在内存中，因为索引往往以索引文件的形式存储在磁盘中。

3）我们平常所说的索引，如果没有特别指明，都是指B树（BTree），（多路搜索树，并不一定是二叉树）结构组织的索引。其中聚集索引，次要索引，复合索引，前缀索引，唯一索引默认都是使用B+树索引，统称索引。当然除了B+树这种类型索引之外还有哈希索引(hash index)等。

2、索引的优势和劣势

- 优势：
  - 查找：类似大学图书馆建书目录索引，提高数据检索的效率，降低数据库的IO成本；
  - 排序：通过索引列对数据进行排序，降低数据排序的成本，降低CPU的消耗。
- 劣势：
  - 实际上索引也是一张表，该表保存了主键与索引字段，并指向实体表的记录，所以索引列也要占用空间的。
  - 虽然索引大大的提供了查询速度，同时会降低更新表的速度，如果对表进行增、删、改操作时，MySQL不仅要更新数据，还要更新一下索引文件。每次更新添加了索引列的字段数据，都会调整因为更新所带来的键值变化后的索引信息。
  - 索引只是提高效率的一个因素，如果你的MySQL有大数据量的表，就需要花时间研究建立最优秀的索引。或者优化查询。（专业的DBA职责）

3、MYSQL索引数据结构

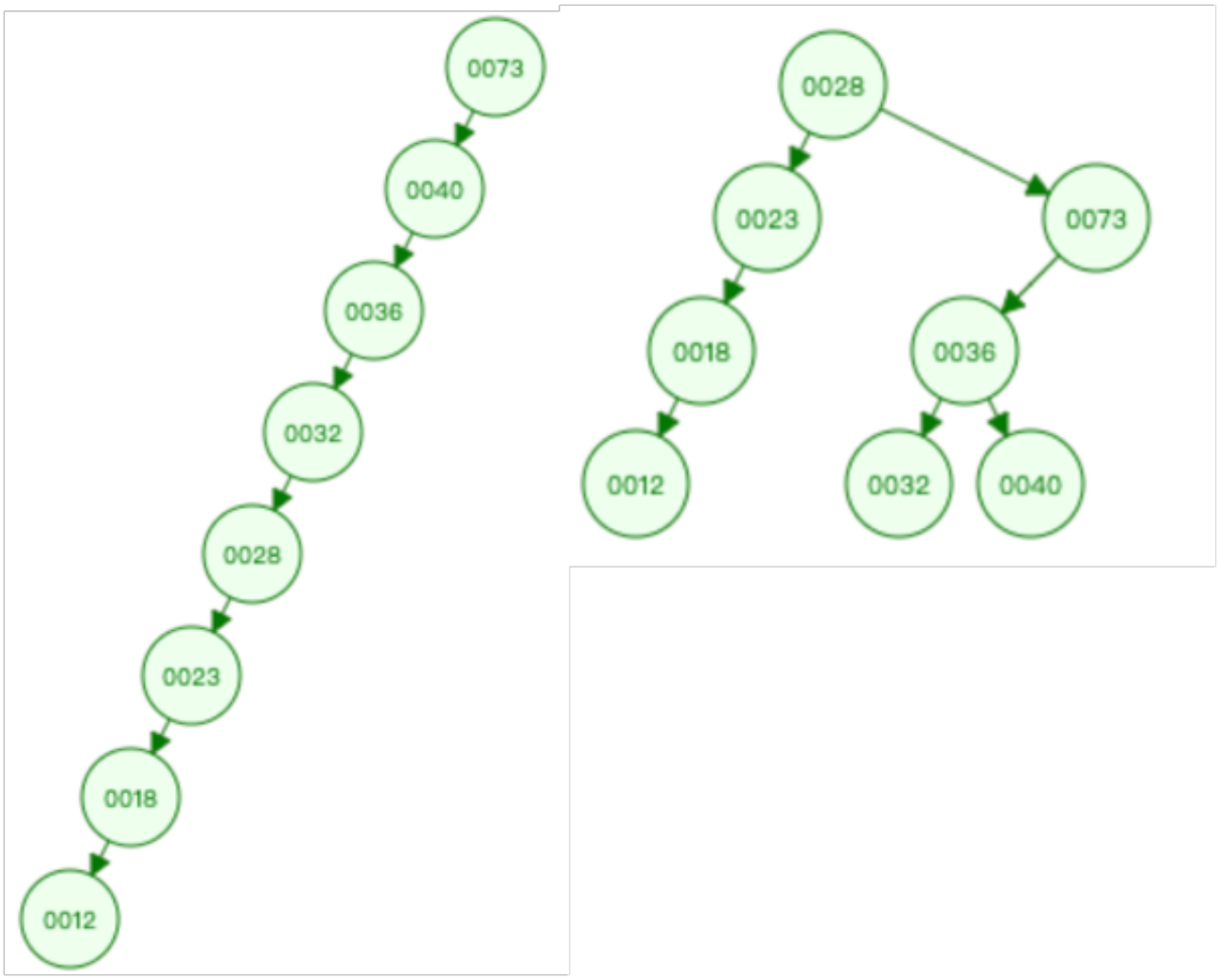
- B+TREE 索引：最常见的索引类型，大部分引擎都支持B+树索引
- HASH索引：底层数据结构是用哈希表实现，只有精确匹配索引列的查询才有效，不支持范围查询
- R-TREE索引：空间索引是 MyISAM 引擎的一个特殊索引类型，主要用于地理空间数据类型，通常使用较少
- FULL Text全文索引：是一种通过建立倒排索引，快速匹配文档的方式，类似于 Lucene, Solr, ES

各存储引擎的支持

索引	InnoDB	MyISAM	Memory
B+Tree索引	支持	支持	支持
Hash索引	不支持	不支持	支持
R-Tree索引	不支持	支持	不支持
Full-text	5.6版本后支持	支持	不支持

https://www.cs.usfca.edu/~galles/visualization/Algorithms.html

1）二叉搜索树

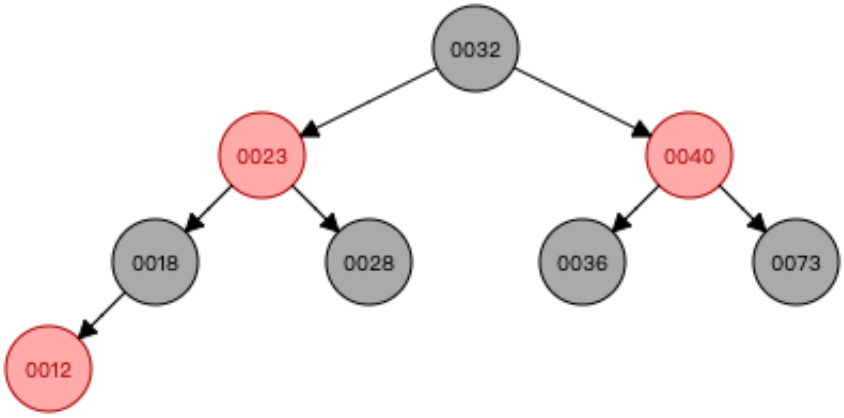


测试数据：28、73、36、23、18、32、12、40

二叉搜索树缺点：顺序插入时，会形成一个链表，查询性能大降低。大数据的情况下，层级较深，检索速度慢。

2）红黑树

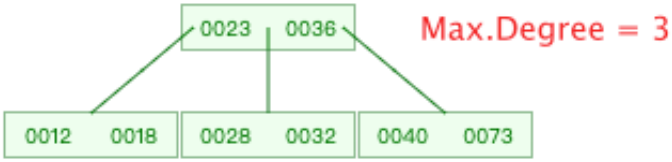
在二叉树的基础上多了树平衡，也叫二叉平衡树，不像二叉树那样极端的情况会往一个方向发展。



红黑树相对二叉排序树而言有所优化，但是同样的也存在大数据的情况下，层级较深，检索速度慢的问题。

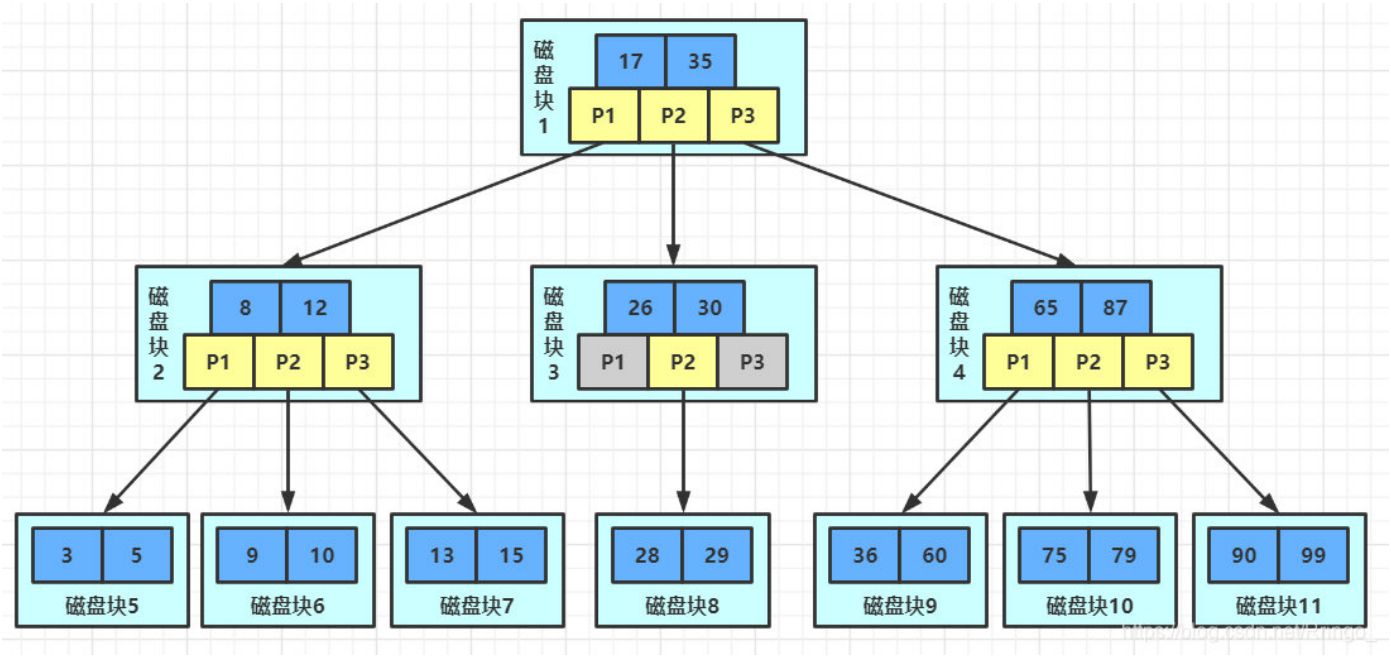
3）B Trees

多路平衡查找树，在红黑树的基础上，每个节点可以存放多个数据。以下是最大度数为3（3阶）的B-TREE。



- 度数：一个节点的子节点个数；
- 最大度数（max-degree），一个节点的最多子节点个数；
- N阶B树，最大度数为N，
  - 每个节点最多可以存储N-1个key（关键字）
  - 每个节点最多有N+1个指针
- 原理
  - 左子树 < 根 < 右子树
  - 插入关键字时，如果节点已满，则将其中间关键字分裂成两个结点，中间关键字被提升到该结点的父结点
- 在B树中，具体的数据都挂载在KEY下面，如下图所示：

思考：如果插入81，上图会发生什么变化？



B Trees的特性：

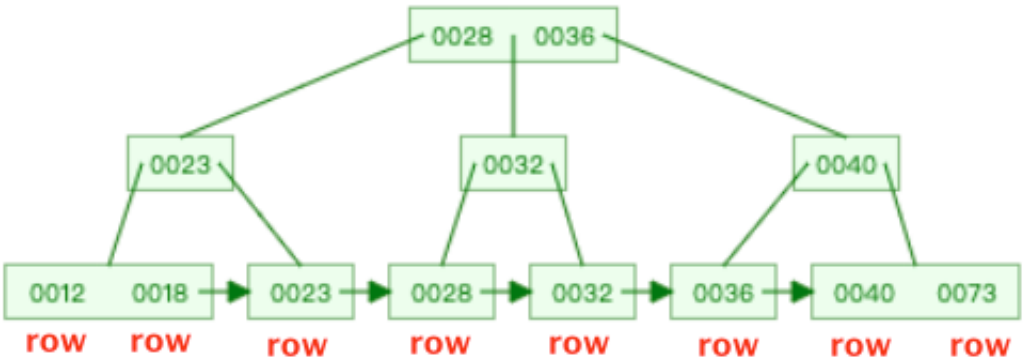
- 关键字集合分布在整颗树中；
- 任何一个关键字出现且只出现在一个结点中；
- 搜索有可能在非叶子结点结束；
- 其搜索性能等价于在关键字全集内做一次二分查找；
- 自动层次控制；

B Trees的搜索，从根结点开始，对结点内的关键字（有序）序列进行二分查找，如果命中则结束，否则进入查询关键字所属范围的儿子结点；重复，直到所对应的儿子指针为空，或已经是叶子结点；

4）B+ Trees

B Trees树是B Trees的变体，也是一种多路搜索树。与B Trees的区别：

- B+ Trees只会在叶子节点上面挂载数据，而非叶子节点不会存放数据，只存放索引列的数据；
- 叶子节点形成一个意向链表



B+的特性：

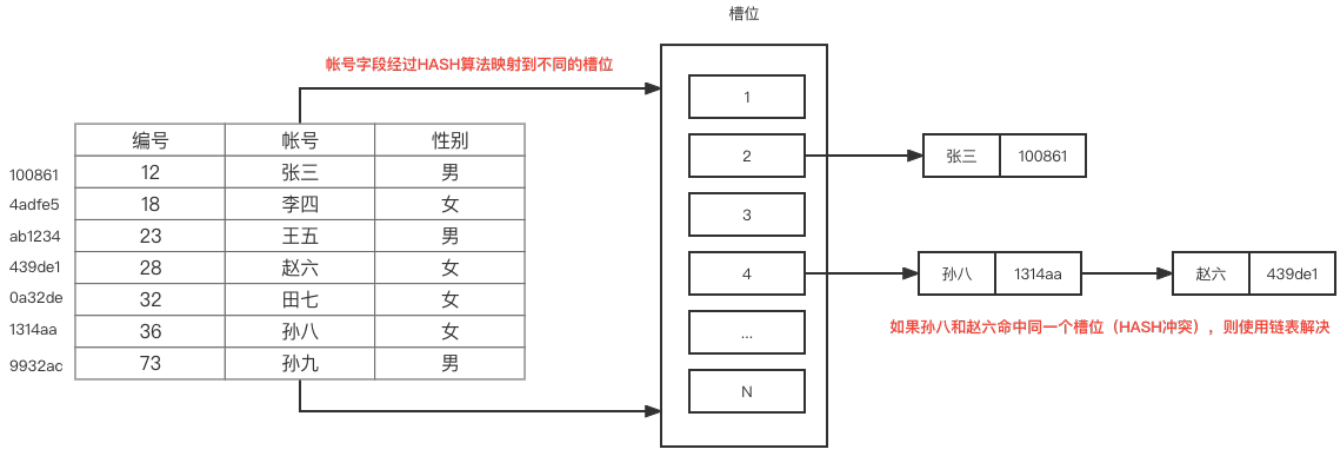
- 所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的；
- 不可能在非叶子结点命中；
- 非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层；
- 更适合文件索引系统；

B+ Trees的搜索与B Trees基本相同，区别是B+Trees只有达到叶子结点才命中（B Trees可以在非叶子结点命中），其性能也等价于在关键字全集做一次二分查找；

在MySQL中，对B+ Trees进行了优化，在原B+ Trees的基础上，增加一个指向相依叶子节点的链表指针。也就是说，叶子节点之间是双向指针连接，从而提高区间范围性能，范围查找。

5）Hash索引

哈希索引就是采用一定的hash算法，将键值换算成新的hash值，映射到对应的槽位上，然后存储在hash表中。 如果两个（或多个）键值，映射到一个相同的槽位上，他们就产生了hash冲突（也称为hash碰撞），可以通过链表来解决。



特点：

- Hash索引只能用于对等比较（=、in），不支持范围查询（betwwn、>、<、...）
- 无法利用索引完成排序操作
- 查询效率高，通常只需要一次检索就可以了，效率通常要高于 B+Tree 索引

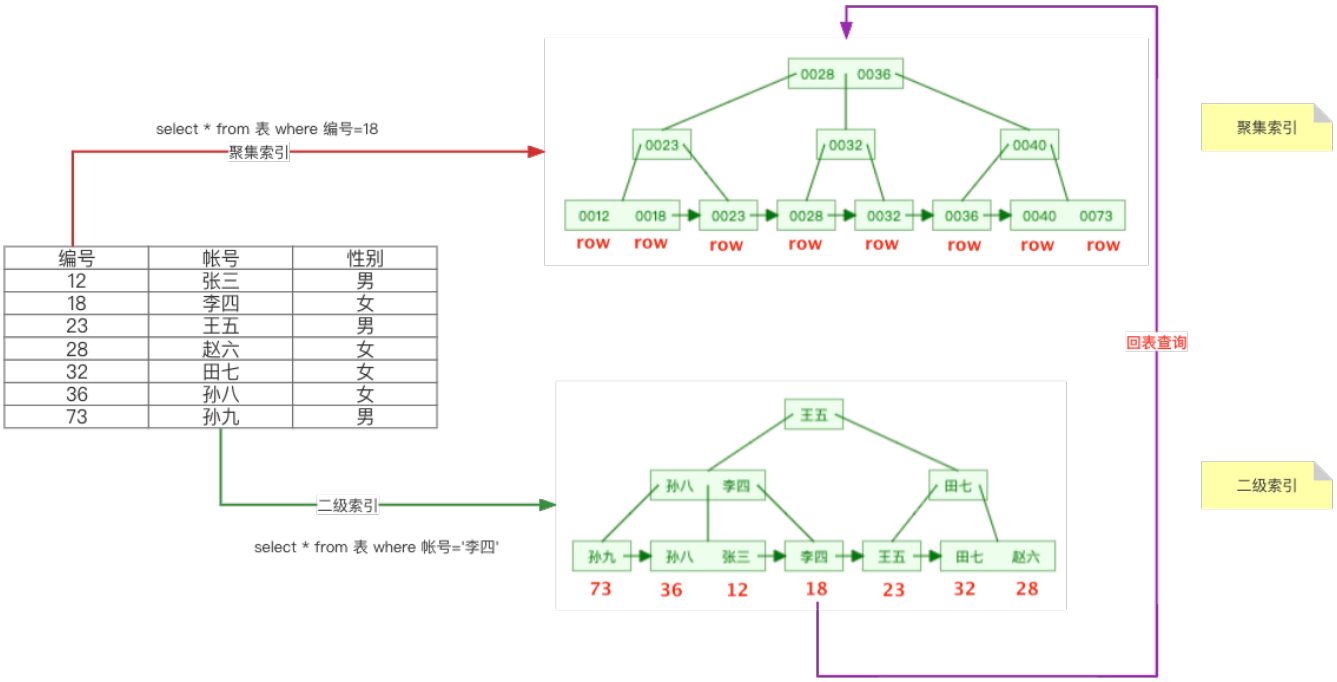
4、索引分类

- 单值索引：一个索引只包含单个列，一个表可以有多个单列索引（建议不超过5个索引）
- 复合索引：一个索引包含多个字段。（如帐号+密码两个字段建立一个复合索引）
- 主键索引（PRIMARY）：表中主键创建的索引，只能有一个，不允许为NULL。（创建主键索引时自动创建）
- 唯一索引（UNIQUE）：索引值的列必须唯一，可以有多个，但允许为NULL。（创建唯一索引时自动创建）

- 全文索引（FULLTEXT）：全文索引查找的是文本中的关键词，而不是比较索引中的值，可以有多个

在 InnoDB 存储引擎中，根据索引的存储形式，又可以分为以下两种：

分类	含义	特点
聚集索引（Clustered Index）	将数据存储与索引放一块，索引结构的叶子节点保存了行数据	必须有，而且只有一个
二级索引（Secondary Index）	将数据与索引分开存储，索引结构的叶子节点关联的是对应的主键	可以存在多个



说明：

- 二级索引，也叫非聚集索引；
- 为什么推荐InnoDB表必须有主键?
  - 保证会有主键索引树的存在（因为数据存放在主键索引树上面）
  - 聚集索引选取规则：
    - 如果存在主键，主键索引就是聚集索引
    - 如果不存在主键，将使用第一个唯一(UNIQUE)索引作为聚集索引
    - 如果表没有主键或没有合适的唯一索引，则 InnoDB 会自动生成一个 rowid 作为隐藏的聚集索引
- 为什么非主键索引结构叶子节点存储的是主键值?
  - 一是保证一致性，更新数据的时候只需要更新主键索引树
  - 二是节省存储空间
- 为什么推荐使用整型的自增主键?
  - 一是方便查找比较
  - 二是新增数据的时候只需要在最后加入，不会大规模调整树结构。如果是UUID的话，大小不好比较，新

增的时候也极有可能在中间插入数据，会导致树结构大规调整，造成插入数据变慢

5、什么情况建立索引

- 主键自动建立唯一索引
- 频繁作为查询条件的字段应该建立索引。最好是保存以后不再变更的字段，因为在增、删、改操作时，会造成数据的变动，同时索引文件也会变动。而在删除操作时，会带来索引文件的重新调整。那么我们可以这样避免：在表增加一个状态字段is_delete=0，如果用户删除数据时，把is_delete=1即可。这样可以避免变动索引文件，从而减少资源的消耗。
- 查询中与其他表关联的字段，即外键关系建立索引
- 要控制索引的数量（不要超过5个索引），太多就性能瓶颈和维护成本，影响增删改的效率
- 一般在开发中建议建立复合索引，如以下字段：
    - id name account password email createtime
    - 我们会建立account+password的复合索引，…where account=? and password=?
- 查询中排序的字段，排序字段若通过索引去访问将大大的提供排序速度。
- 查询中统计或者分组字段（group by也和索引有关）。

6、什么情况不建立索引

- 频繁更新的字段不适合作为索引，因为每次更新不单单是更新记录还在更新索引。
- 表的记录数太少
- 经常增删改的表
- 数据重复且平均的字段。
- where字段用不到的字段不要建立索引
- 假如一个表有10万行记录，有一个字段A只有true和false两种值，并且每个值的分布概率大约为50%，那么对A字段建索引一般不会提高数据库的查询速度。索引的选择性是指索引列中不同值的数目与表中记录数的比。如果一个表中有2000条记录，表索引列有1980个不同的值，那么这个索引的选择性就是1980/2000=0.99。一个索引的选择性越接近于1，这个索引的效率就越高。

7、操作索引

1）创建语法一

```
# 索引命名规范：idx_xxx
CREATE [UNIQUE] INDEX 索引名称 ON 表名 (字段|字段列表) ;

# 单值索引
CREATE INDEX 索引名称 ON 表名 (字段) ;
CREATE INDEX index_username ON userinfo(username) ;

# 唯一索引
CREATE UNIQUE INDEX 索引名称 ON 表名 (字段列表) ;
```

```
CREATE UNIQUE INDEX index_username ON userinfo(username) ;

# 复合索引,字段列表中的字段顺序是有讲究的
CREATE INDEX 索引名称 ON 表名(字段列表) ;
CREATE INDEX index_username_password ON userinfo(username,password) ;
```

案例

> 需要引入MySQL示例数据库
>
> ```
> # 下载: https://www.mysql.com/
> DOCUMENTATION - More - Example Databases
> https://github.com/datacharmer/test_db/releases/tag/v1.0.7
>
> # 安装
> mysql < employees.sql -u root -p
>
> # 测试
> mysql -t < test_employees_md5.sql -u root -p
> ```

```
# emp_no是主键，以下查询会使用到主键索引
mysql> select * from employees where emp_no=10001;

# first_name字段并没有添加索引，因此查询速度会很慢
mysql> select * from employees where first_name='Facello';

# 创建索引
create index idex_employees_first_name on employees(first_name);

# 再次查询测试
mysql> select * from employees where first_name='Facello';
```

2）创建语法二

```
ALTER TABLE 表名 ADD [UNIQUE] INDEX 索引名称(字段|字段列表)

# 单值索引
ALTER TABLE 表名 ADD INDEX 索引名称(字段)
ALTER TABLE userinfo ADD INDEX index_username(username)

# 唯一索引
ALTER TABLE 表名 ADD UNIQUE INDEX 索引名称(字段列表)
ALTER TABLE userinfo ADD UNIQUE INDEX index_username ON (username)

# 该语句添加一个主键，这意味着索引值必须是唯一的，并且不能为NULL
ALTER TABLE 表名 ADD INDEX 索引名称(字段列表)
ALTER TABLE userinfo ADD INDEX index_username_password ON (username,password)
```

```
# 主键索引（特殊的唯一索引）
ALTER TABLE 表名 ADD PRIMARY KEY(字段列表);
```

注意：

- 如果某个字段设置为主键约束（primary key），那么该字段默认就是主键索引。
- 主键索引是特殊的唯一索引
    - 相同点：该列中的数据都不能有相同值；
    - 不同点：主键索引不能有null值，但是唯一索引可以有null值。

3）全文检索

```
# 该语句指定了索引为FULLTEXT，用于全文检索
ALTER TABLE 表名 ADD FULLTEXT 索引名称(字段列表);
```

4）查看索引

```
SHOW INDEX FROM 表名; \G
```

5）删除索引

```
DROP INDEX 索引名称 ON 表名 ;
```

三、性能分析

1、查看执行频次

查看当前数据库的 INSERT, UPDATE, DELETE, SELECT 访问频次：

```
# GLOBAL关键字表示全局系统变量，SESSION关键字表示会话系统变量，如果不指定关键字，默认就是会话系统变量。
SHOW [GLOBAL | SESSION] STATUS LIKE 'Com_______';

SHOW GLOBAL STATUS LIKE 'Com_______';
```

2、慢查询日志

慢查询日志记录了所有执行时间超过指定参数（long_query_time，单位：秒，默认10秒）的所有SQL语句的日志。

1）查看慢查询日志开关状态

```
mysql> show variables like 'slow_query_log';
+----------------+-------+
| Variable_name  | Value |
+----------------+-------+
| slow_query_log | OFF   |
+----------------+-------+
1 row in set, 1 warning (0.02 sec)
```

2）查看慢查询相关的参数

```
mysql> show variables like '%query%';
+------------------------------+-----------------------------------------------+
| Variable_name                | Value                                         |
+------------------------------+-----------------------------------------------+
| binlog_rows_query_log_events | OFF                                           |
| ft_query_expansion_limit     | 20                                            |
| have_query_cache             | YES                                           |
| long_query_time              | 2.000000                                      |
| query_alloc_block_size       | 8192                                          |
| query_cache_limit            | 1048576                                       |
| query_cache_min_res_unit     | 4096                                          |
| query_cache_size             | 1048576                                       |
| query_cache_type             | OFF                                           |
| query_cache_wlock_invalidate | OFF                                           |
| query_prealloc_size          | 8192                                          |
| slow_query_log               | ON                                            |
| slow_query_log_file          | C:\mysql-5.7.32-winx64\logs\localhost-slow.log |
+------------------------------+-----------------------------------------------+
13 rows in set, 1 warning (0.00 sec)
```

- long_query_time：慢查询时间
- slow_query_log：慢查询开启状态
- slow_query_log_file：慢查询日志文件位置

3）慢查询配置

MySQL的慢查询日志默认没有开启，需要在MySQL的配置文件中配置如下信息：

- windows系统
  - 配置文件：`my.ini`
  - 默认慢查询日志位置：`data目录/主机名-slow.log`
- linux系统

- 配置文件：`/etc/my.cnf`
- 默认慢查询日志位置：`/var/lib/mysql/localhost-slow.log`

```
# 慢日志路径
 slow_query_log_file=C:\mysql-5.7.32-winx64\logs\localhost-slow.log

# 开启慢查询日志开关
slow_query_log=1

# 设置慢查询日志的时间为2秒，SQL语句执行时间超过2秒，就会视为慢查询，记录慢查询日志
long_query_time=2
```

修改配置后，重启MySQL服务使其生效

查询测试

```
mysql> select * from titles,employees limit 20000000,10;
...
10 rows in set (3.14 sec)
```

查看日志文件

```
MySQL5.7.32, Version: 5.7.32-log (MySQL Community Server (GPL)). started with:
TCP Port: 3306, Named Pipe: (null)
Time                 Id Command    Argument
# Time: 2022-04-09T06:33:01.113650Z
# User@Host: root[root] @ localhost [::1]  Id:     2
# Query_time: 3.134978  Lock_time: 0.004415 Rows_sent: 10  Rows_examined: 11937
use employees;
SET timestamp=1649485981;
select * from titles,employees limit 20000000,10;
```

3、profile

show profile 能在做SQL优化时帮我们了解时间都耗费在哪里。

1）查询是否支持

```
# 通过 have_profiling 变量，查看当前 MySQL 是否支持
mysql> select @@have_profiling;
+------------------+
| @@have_profiling |
+------------------+
| YES              |
+------------------+
1 row in set, 1 warning (0.00 sec)
```

2）开启profile

profiling 默认关闭，可以通过以下语句开启：

```
#语法：  SET [GLOBAL | SESSION] profiling = 1 | 0 ;


# 当前会话中，开启profiling
SET profiling = 1

# 查看是否开启
select @@profiling;
```

3）查看所有语句的耗时

```
mysql> show profiles;
+----------+------------+-------------------------------------------------+
| Query_ID | Duration   | Query                                           |
+----------+------------+-------------------------------------------------+
|        1 | 0.00021825 | select @@profiling                              |
|        2 | 3.18666900 | select * from titles,employees limit 20000000,10 |
+----------+------------+-------------------------------------------------+
2 rows in set, 1 warning (0.00 sec)
```

4）查看指定Query_ID的SQL语句各个阶段的耗时

```
#语法：  show profile for query query_id;

mysql> show profile for query 2;
+----------------------+----------+
| Status               | Duration |
+----------------------+----------+
| starting             | 0.000071 |
| checking permissions | 0.000004 |
| checking permissions | 0.000004 |
| Opening tables       | 0.000034 |
| init                 | 0.000036 |
| System lock          | 0.000010 |
| optimizing           | 0.000003 |
| statistics           | 0.000020 |
| preparing            | 0.000020 |
| executing            | 0.000002 |
| Sending data         | 3.185604 |
| end                  | 0.000016 |
| query end            | 0.000015 |
| closing tables       | 0.000011 |
```

```
| freeing items        | 0.000108 |
| logging slow query   | 0.000687 |
| cleaning up          | 0.000026 |
+----------------------+----------+
17 rows in set, 1 warning (0.00 sec)
```

5）查看指定query_id的SQL语句CPU的使用情况

```
# 语法：  show profile cpu for query query_id;

mysql> show profile cpu for query 2;
+----------------------+----------+----------+------------+
| Status               | Duration | CPU_user | CPU_system |
+----------------------+----------+----------+------------+
| starting             | 0.000071 | 0.000000 |   0.000000 |
| checking permissions | 0.000004 | 0.000000 |   0.000000 |
| checking permissions | 0.000004 | 0.000000 |   0.000000 |
| Opening tables       | 0.000034 | 0.000000 |   0.000000 |
| init                 | 0.000036 | 0.000000 |   0.000000 |
| System lock          | 0.000010 | 0.000000 |   0.000000 |
| optimizing           | 0.000003 | 0.000000 |   0.000000 |
| statistics           | 0.000020 | 0.000000 |   0.000000 |
| preparing            | 0.000020 | 0.000000 |   0.000000 |
| executing            | 0.000002 | 0.000000 |   0.000000 |
| Sending data         | 3.185604 | 3.125000 |   0.000000 |
| end                  | 0.000016 | 0.000000 |   0.000000 |
| query end            | 0.000015 | 0.000000 |   0.000000 |
| closing tables       | 0.000011 | 0.000000 |   0.000000 |
| freeing items        | 0.000108 | 0.000000 |   0.000000 |
| logging slow query   | 0.000687 | 0.000000 |   0.000000 |
| cleaning up          | 0.000026 | 0.000000 |   0.000000 |
+----------------------+----------+----------+------------+
17 rows in set, 1 warning (0.00 sec)
```

4、Explain的简介

EXPLAIN：SQL的执行计划，使用EXPLAIN关键字可以模拟优化器执行SQL查询语句，从而知道MySQL是如何处理SQL语句的。语法如下所示：

```
desc | explain SQL语句
```

1）EXPLAIN字段

```
# 创建数据库
CREATE DATABASE if not exists explaintest DEFAULT character set = utf8 ;
use explaintest ;

# 用户信息表
CREATE TABLE userinfo
(
  `id` int  ,
  `username` varchar(50)  ,
  `password` varchar(50) ,
  `sex` char(2),
  `age` tinyint
);

INSERT into userinfo (id,username,password,sex,age) VALUES (1001,'zs','111111','男',18) ;
INSERT into userinfo (id,username,password,sex,age) VALUES (1002,'ls','222222','女',28) ;
INSERT into userinfo (id,username,password,sex,age) VALUES (1003,'ww','333333','男',38) ;


# 订单信息表
CREATE TABLE orderinfo
(
  `id` int  ,
  `total` DECIMAL(10,1)  ,
  `date` datetime,
  `userid` int
);

INSERT into orderinfo(id,total,date,userid) values (11,55.4,'2022-02-21 11:11:11.111',1001) ;
INSERT into orderinfo(id,total,date,userid) values (12,100,'2022-02-22 22:22:22.222',1002) ;
INSERT into orderinfo(id,total,date,userid) values (13,136.6,'2022-02-23 12:12:12.122',1003) ;


# 地址信息表
create TABLE address
(
  `id` int  ,
  `detail` varchar(100)  ,
  `isdefault` bool,
  `userid` int
);

insert into address(id,detail,isdefault,userid) values (21,'广东珠海',1,1001) ;
insert into address(id,detail,isdefault,userid) values (22,'广东广州',1,1002) ;
insert into address(id,detail,isdefault,userid) values (23,'广东中山',1,1003) ;
insert into address(id,detail,isdefault,userid) values (24,'广东深圳',0,1003) ;
```

```
mysql> explain select * from userinfo \G
```

```
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: userinfo
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 3
     filtered: 100.00
        Extra: NULL
1 row in set, 1 warning (0.00 sec)
```

- id：编号，表的读取和加载顺序
- select_type：查询类型
- table：表
- type：类型
- possible_keys：预测可能用到的索引，一个或多个
- key：实际使用的索引，如果为NULL，则表示没有使用索引
- key_len：实际使用索引的长度
- ref：表之间的引用
- rows：通过索引查询到的数据量
- filtered：表示返回结果的行数占需读取行数的百分比，filtered的值越大越好
- Extra：额外的信息

1）id

`id`：表的读取和加载顺序。

值有以下三种情况：

- `id` 相同，执行顺序由上至下。
- `id` 不同，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` 相同不同，同时存在。**永远是id大的优先级最高，id相等的时候顺序执行。**

```
# 案例一：id值相同，数据表按顺序查询
EXPLAIN SELECT * from userinfo t1,orderinfo t2 ,address t3
where t1.id=t2.userid and t1.id = t3.userid;
```

Message | Result 1 | Profile | Status

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t1	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	3	100.00	(NULL)
1	SIMPLE	t2	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	3	33.33	Using where; Using join buffer (Block Nested Loop)
1	SIMPLE	t3	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	4	25.00	Using where; Using join buffer (Block Nested Loop)

说明：以上操作进行了三个简单查询，由于id值相同，因此按顺序执行查询，分别为：t1、t2、t3

```
# 案例二：表的执行顺序会因表数量的改变而改变
# 在userinfo表，添加三条新的数据，再执行以上同样的测试
INSERT into userinfo (id,username,password,sex,age) VALUES (1004,'zl','444444','女',31) ;
INSERT into userinfo (id,username,password,sex,age) VALUES (1005,'tq','555555','女',21) ;
INSERT into userinfo (id,username,password,sex,age) VALUES (1006,'sb','666666','男',22) ;

EXPLAIN SELECT * from userinfo t1,orderinfo t2 ,address t3
where t1.id=t2.userid and t1.id = t3.userid;
```

Message | Result 1 | Profile | Status

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t2	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	3	100.00	(NULL)
1	SIMPLE	t3	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	4	25.00	Using where; Using join buffer (Block Nested Loop)
1	SIMPLE	t1	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	6	16.67	Using where; Using join buffer (Block Nested Loop)

说明：id值相同，仍然是从上往下顺序执行。但发现表的查询顺序不一样了。表的执行顺序会因表数量的改变而改变。原因：笛卡尔积。分析：

```
t1 t2 t3
6  3  4
最终：  6 * 3 * 4 = 18 * 4 = 72

t2 t3 t1
3  4  6
最终：  3 * 4 * 6 = 12 * 6 = 72
```

最终执行的条数，虽然是一致的（72条）。但是中间过程，有一张临时表是18，一张临时表是12，很明显12 < 18，对于内存来说，数据量越小越好，因此优化器肯定会选择第二种执行顺序。

```
# 案例三：id值不同（嵌套子查询）
EXPLAIN select * from orderinfo t1
where t1.userid =
(
  select t2.id from userinfo t2 WHERE t2.id=
  (
    SELECT t3.userid from address t3 where id=23
  )
);
```

Message | Result 1 | Profile | Status

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	t1	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	3	33.33	Using where
2	SUBQUERY	t2	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	6	16.67	Using where
3	SUBQUERY	t3	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	4	25.00	Using where

说明：id值不同，id值越大越优先查询。这是由于在进行嵌套子查询时，先查内层，再查外层。

```
# 案例四：id值同时存在相同和不相同
EXPLAIN select * from orderinfo t1 ,userinfo t2
where t1.userid = t2.id and t1.userid=
(
    SELECT t3.userid from address t3 where id=23
);
```

Message | Result 1 | Profile | Status

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	t1	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	3	33.33	Using where
1	PRIMARY	t2	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	6	16.67	Using where
2	SUBQUERY	t3	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	4	25.00	Using where

说明：id值有相同，也有不同。id值越大越优先；id值相同，从上往下顺序执行。

2）select_type

`select_type`：数据查询的类型，主要是用于区别，普通查询、联合查询、子查询等的复杂查询。

- `SIMPLE`：简单的`SELECT`查询，查询中不包含子查询或者`UNION`。
- `PRIMARY`：主查询，查询中如果包含任何复杂的子部分，最外层查询则被标记为`PRIMARY`。
- `SUBQUERY`：子查询，在`SELECT`或者`WHERE`子句中包含了子查询。
- `DERIVED`：衍生查询，在`FROM`子句中包含的子查询被标记为`DERIVED(衍生)`，MySQL会递归执行这些子查询，把结果放在临时表中。
- `UNION`：如果第二个`SELECT`出现在`UNION`之后，则被标记为`UNION`；若`UNION`包含在`FROM`子句的子查询中，外层`SELECT`将被标记为`DERIVED`。
- `UNION RESULT`：从`UNION`表获取结果的`SELECT`。

```
explain SELECT * from (
  SELECT * from address t1 where t1.isdefault = 1          # union之前，select_type标记为：derived
  union
  SELECT * from address t2 where t2.isdefault = 0          # union之后，select_type标记为：union
) as t3 ;
```

Message | Result 1 | Profile | Status

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	<derived2>	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	4	100.00	(NULL)
2	DERIVED	t1	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	4	25.00	Using where
3	UNION	t2	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	4	25.00	Using where
(NULL)	UNION RESULT	<union2,3>	(NULL)	ALL	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	(NULL)	Using temporary

说明：在上面的查询SQL语句中，`union`关键字之后的表（t2）标记为union，而`union`关键字之前的表标记为derived。`union result`：告诉我们，哪些表之间使用了union查询。

3）type

`type`：访问类型排列。

从最好到最差依次是： `system` > `const` > `eq_ref` > `ref` > `range` > `index` > `ALL`。除了 `ALL` 没有用到索引，其他级别都用到索引了。`system`、`const` 只是理想状况，实际上只能优化到 `index`、`range`、`ref` 这些级别。一般来说，得保证查询至少达到 `range` 级别，最好达到 `ref`。

- `system`：表只有一行记录（等于系统表），这是 `const` 类型的特例，平时不会出现，这个也可以忽略不计。
- `const`：表示通过索引一次就找到了，`const` 用于比较 `primary key` 或者 `unique` 索引。因为只匹配一行数据，所以很快。如将主键置于 `where` 列表中，MySQL就能将该查询转化为一个常量。
- `eq_ref`：唯一性索引扫描，读取本表中和关联表表中的每行组合成的一行，查出来只有一条记录。除了 `system` 和 `const` 类型之外，这是最好的联接类型。
- `ref`：非唯一性索引扫描，返回本表和关联表某个值匹配的所有行，查出来有多条记录。
- `range`：只检索给定范围的行，一般就是在 `WHERE` 语句中出现了 `BETWEEN`、`< >`、`in` 等的查询。这种范围扫描索引比全表扫描要好，因为它只需要开始于索引树的某一点，而结束于另一点，不用扫描全部索引。
- `index`：`Full Index Scan`，全索引扫描，`index` 和 `ALL` 的区别为 `index` 类型只遍历索引树。**也就是说虽然 `ALL` 和 `index` 都是读全表，但是 `index` 是从索引中读的，`ALL` 是从磁盘中读取的。**
- `ALL`：`Full Table Scan`，没有用到索引，全表扫描。

4）possible_keys 和 key

`possible_keys`：显示可能应用在这张表中的索引，一个或者多个。查询涉及到的字段上若存在索引，则该索引将被列出，**但不一定被查询实际使用。**

`key`：实际使用的索引。如果为 `NULL`，则没有使用索引。查询中如果使用了覆盖索引，则该索引仅仅出现在 `key` 列表中。

5）key_len

`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。在不损失精度的情况下，长度越短越好。

`key_len` 计算规则：https://blog.csdn.net/qq_34930488/article/details/102931490

```
mysql> desc pms_category;
+---------------+------------+------+-----+---------+----------------+
| Field         | Type       | Null | Key | Default | Extra          |
+---------------+------------+------+-----+---------+----------------+
| cat_id        | bigint(20) | NO   | PRI | NULL    | auto_increment |
| name          | char(50)   | YES  |     | NULL    |                |
| parent_cid    | bigint(20) | YES  |     | NULL    |                |
| cat_level     | int(11)    | YES  |     | NULL    |                |
| show_status   | tinyint(4) | YES  |     | NULL    |                |
| sort          | int(11)    | YES  |     | NULL    |                |
```

```
| icon          | char(255)  | YES  |     | NULL    |                |
| product_unit  | char(50)   | YES  |     | NULL    |                |
| product_count | int(11)    | YES  |     | NULL    |                |
+---------------+------------+------+-----+---------+----------------+
9 rows in set (0.00 sec)


mysql> explain select cat_id from pms_category where cat_id between 10 and 20 \G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: pms_category
   partitions: NULL
         type: range
possible_keys: PRIMARY
          key: PRIMARY  # 用到了主键索引，通过查看表结构知道，cat_id是bigint类型，占用8个字节
      key_len: 8        # 这里只用到了cat_id主键索引，所以长度就是8!
          ref: NULL
         rows: 11
     filtered: 100.00
        Extra: Using where; Using index
1 row in set, 1 warning (0.00 sec)
```

6）ref

`ref`：显示索引的哪一列被使用了，如果可能的话，是一个常数。哪些列或常量被用于查找索引列上的值。

7）rows

`rows`：根据表统计信息及索引选用情况，大致估算出找到所需的记录需要读取的行数。

8）Extra

`Extra`：包含不适合在其他列中显示但十分重要的额外信息。

- `Using filesort`：说明MySQL会对数据使用一个外部的索引排序，而不是按照表内的索引顺序进行读取。**MySQL中无法利用索引完成的排序操作成为"文件内排序"。**

```
# 排序没有使用索引
mysql> explain select name from pms_category where name='Tangs' order by cat_level \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: pms_category
   partitions: NULL
         type: ref
possible_keys: idx_name_parentCid_catLevel
          key: idx_name_parentCid_catLevel
```

```
      key_len: 201
          ref: const
         rows: 1
     filtered: 100.00
        Extra: Using where; Using index; Using filesort
1 row in set, 1 warning (0.00 sec)

#~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
# 排序使用到了索引

mysql> explain select name from pms_category where name='Tangs' order by parent_cid,cat_level\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: pms_category
   partitions: NULL
         type: ref
possible_keys: idx_name_parentCid_catLevel
          key: idx_name_parentCid_catLevel
      key_len: 201
          ref: const
         rows: 1
     filtered: 100.00
        Extra: Using where; Using index
1 row in set, 1 warning (0.00 sec)
```

- `Using temporary`：使用了临时表保存中间结果，MySQL在对查询结果排序时使用了临时表。常见于排序 `order by` 和分组查询 `group by`。**临时表对系统性能损耗很大。**
- `Using index`：表示相应的 `SELECT` 操作中使用了覆盖索引，避免访问了表的数据行，效率不错！如果同时出现 `Using where`，表示索引被用来执行索引键值的查找；如果没有同时出现 `Using where`，表明索引用来读取数据而非执行查找动作。

```
# 覆盖索引
# 就是select的数据列只用从索引中就能够取得，不必从数据表中读取，换句话说查询列要被所使用的索引覆盖。
# 注意：如果要使用覆盖索引，一定不能写SELECT *，要写出具体的字段。
mysql> explain select cat_id from pms_category \G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: pms_category
   partitions: NULL
         type: index
possible_keys: NULL
          key: PRIMARY
      key_len: 8
          ref: NULL
         rows: 1425
     filtered: 100.00
        Extra: Using index   # select的数据列只用从索引中就能够取得，不必从数据表中读取
1 row in set, 1 warning (0.00 sec)
```

- `Using where`：表明使用了 `WHERE` 过滤。
- `Using join buffer`：使用了连接缓存。
- `impossible where`： `WHERE` 子句的值总是false，不能用来获取任何元组。

```
mysql> explain select name from pms_category where name = 'zs' and name = 'ls'\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: NULL
   partitions: NULL
         type: NULL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
     filtered: NULL
        Extra: Impossible WHERE   # 不可能字段同时查到两个名字
1 row in set, 1 warning (0.00 sec)
```

2）索引分析

```
create table stu
(
    id int auto_increment primary key ,
    name varchar(50),
    sex int default 1,
    age int,
    weight float
);

insert into stu (name,age,weight) values ('z1',2,50.5) ;
insert into stu (name,age,weight) values ('ls',3,50.5) ;
insert into stu (name,age,weight) values ('ww',4,60.5) ;
insert into stu (name,age,weight) values ('zl',5,51.5) ;
insert into stu (name,age,weight) values ('tq',6,55) ;

# 查看索引
mysql> show index from stu\G;
*************************** 1. row ***************************
        Table: stu
   Non_unique: 0
     Key_name: PRIMARY
 Seq_in_index: 1
  Column_name: id
    Collation: A
  Cardinality: 5
     Sub_part: NULL
       Packed: NULL
```

```
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
1 row in set (0.00 sec)

# 性能分析
mysql> explain select id,name from stu where sex=1 and age>4 order by weight limit 1\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: stu
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 5
     filtered: 20.00
        Extra: Using where; Using filesort
1 row in set, 1 warning (0.00 sec)

# 注：如果type为All，是最坏的情况。如果Extra还出现了Using filesort也是最坏的情况，那么就必须进行优化

# 创建索引
create index idx_stu_saw on stu(sex,age,weight);

# 再次查看索引
mysql> show index from stu \G;
*************************** 1. row ***************************
        Table: stu
   Non_unique: 0
     Key_name: PRIMARY
 Seq_in_index: 1
  Column_name: id
    Collation: A
  Cardinality: 5
     Sub_part: NULL
       Packed: NULL
         Null:
   Index_type: BTREE
      Comment:
Index_comment:
*************************** 2. row ***************************
        Table: stu
   Non_unique: 1
     Key_name: idx_stu_saw
 Seq_in_index: 1
  Column_name: sex
    Collation: A
  Cardinality: 1
```

```
     Sub_part: NULL
       Packed: NULL
         Null: YES
   Index_type: BTREE
      Comment:
Index_comment:
*************************** 3. row ***************************
        Table: stu
   Non_unique: 1
     Key_name: idx_stu_saw
 Seq_in_index: 2
  Column_name: age
    Collation: A
  Cardinality: 5
     Sub_part: NULL
       Packed: NULL
         Null: YES
   Index_type: BTREE
      Comment:
Index_comment:
*************************** 4. row ***************************
        Table: stu
   Non_unique: 1
     Key_name: idx_stu_saw
 Seq_in_index: 3
  Column_name: weight
    Collation: A
  Cardinality: 5
     Sub_part: NULL
       Packed: NULL
         Null: YES
   Index_type: BTREE
      Comment:
Index_comment:
4 rows in set (0.00 sec)


# 添加索引后，再测试，发现type的值变为range，性能比之前要好一些，是可以忍受的
# 但，Extra为Using filesort是无法接受的
mysql> explain select id,name from stu where sex=1 and age>4 order by weight limit 1\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: stu
   partitions: NULL
         type: range
possible_keys: idx_stu_saw
          key: idx_stu_saw
      key_len: 10
          ref: NULL
         rows: 2
     filtered: 100.00
```

```
        Extra: Using index condition; Using filesort
1 row in set, 1 warning (0.00 sec)
# 说明：type为range,性能有所提高，但因为范围查询（age>3）导致了索引失效。
# 此时，可以和运维、项目经理、需要经理进行协商，是否必须使用此范围查询。


# 修改范围条件后再测试
# type为ref，表示已经使用了索引
mysql> explain select id,name from stu where sex=1 and age=4 order by weight limit 1\G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: stu
   partitions: NULL
         type: ref
possible_keys: idx_stu_saw
          key: idx_stu_saw
      key_len: 10
          ref: const,const
         rows: 1
     filtered: 100.00
        Extra: Using index condition
1 row in set, 1 warning (0.00 sec)


# 结论：
#####
type为range时，性能有所提高，这是可以忍受的，但Extra为Using filesort是无法接受的。
而我们已经建立了索引，为什么没用上呢?
这是因为BTree索引的工作原理导致的。
先排序sex
如果遇到相同的sex，则在排序age,如果遇到相同的age则再排序weight。
当age字段在联合索引里处于中间位置时，因为age>4,条件是一个范围值，即所谓的range。
MySQL无法利用索引在对后面的weight部分进行检索，即range类型查询字段后面的索引无效。
#####

# 注意：模糊查询、范围查询、in查询等索引无效，都进行全表查询。
```

四、索引失效情况

- 全值匹配我最爱。
- 最佳左前缀法则。
- 不在索引列上做任何操作（计算、函数、(自动or手动)类型转换），会导致索引失效而转向全表扫描。
- 索引中范围条件右边的字段会全部失效。
- 尽量使用覆盖索引（只访问索引的查询，索引列和查询列一致），减少 `SELECT *` 。
- MySQL在使用 `!=` 或者 `<>` 的时候无法使用索引会导致全表扫描。
- `is null` 、 `is not null` 也无法使用索引。

- `like` 以通配符开头 `%abc` 索引失效会变成全表扫描。
- 字符串不加单引号索引失效。
- 少用 `or` ，用它来连接时会索引失效。

```
# 准备数据
CREATE TABLE `staffs`(
  `id` INT(10) PRIMARY KEY AUTO_INCREMENT,
  `name` VARCHAR(24) NOT NULL DEFAULT '' COMMENT '姓名',
  `age` INT(10) NOT NULL DEFAULT 0 COMMENT '年龄',
  `phone` CHAR(11) NOT NULL DEFAULT '' COMMENT '联系方式',
  `pos` VARCHAR(20) NOT NULL DEFAULT '' COMMENT '职位',
  `add_time` TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '入职时间'
) COMMENT '员工记录表';

INSERT INTO `staffs`(`name`,`age`,`phone`,`pos`) VALUES
('Rose',18,'13417740001','dev'),
('Lucy',19,'13417740002','dev'),
('Lily',17,'13417740003','dev'),
('Petter',16,'13417740004','dev'),
('Preusig',15,'13417740005','dev'),
('Zielinski',13,'13417740006','dev'),
('Kalloufi',14,'13417740007','dev'),
('Peac',16,'13417740008','dev'),
('Piveteau',29,'13417740009','dev'),
('Sluis',16,'13417740010','dev'),
('Bridgland',18,'13417740011','dev'),
('Terkki',21,'13417740012','dev'),
('Genin',20,'13417740013','dev'),
('Nooteboom',12,'13417740014','dev'),
('Cappelletti',14,'13417740015','dev'),
('Boulоucos',13,'13417740016','dev'),
('Peha',14,'13417740017','dev'),
('Haddadi',15,'13417740018','dev'),
('Warwick',16,'13417740019','dev');

# 创建复合索引
CREATE INDEX idx_staffs_name_age_pos ON `staffs`(`name`,`age`,`pos`);
```

1）最佳左前缀法则

最佳左前缀法则

- 如果索引是多字段的复合索引，要遵守最佳左前缀法则。指的是查询从索引的最左前列开始并且不跳过索引中的字段；
- 如果跳过第个索引字段，则索引失败；
- 如果跳过某个索引字段，则索引将部分失效；

```
# 用到了idx_staffs_name_age_pos索引中的name字段
EXPLAIN SELECT * FROM `staffs` WHERE `name` = 'Ringo';
```

```
# 用到了idx_staffs_name_age_pos索引中的name，age字段
EXPLAIN SELECT * FROM `staffs` WHERE `name` = 'Ringo' AND `age` = 18;

# 用到了idx_staffs_name_age_pos索引中的name，age，pos字段，这是属于全值匹配的情况！！！
EXPLAIN SELECT * FROM `staffs` WHERE `name` = 'Ringo' AND `age` = 18 AND `pos` = 'manager';

# 同上，正常的使用了复合索引，条件顺序无关
EXPLAIN SELECT * FROM `staffs` WHERE `age` = 18 AND `pos` = 'manager' AND `name` = 'Ringo' ;

# 索引没用上，ALL全表扫描
EXPLAIN SELECT * FROM `staffs` WHERE `age` = 18 AND `pos` = 'manager';

# 索引没用上，ALL全表扫描
EXPLAIN SELECT * FROM `staffs` WHERE `pos` = 'manager';

# 用到了idx_staffs_name_age_pos索引中的name字段，pos字段索引失效
EXPLAIN SELECT * FROM `staffs` WHERE `name` = 'Ringo' AND `pos` = 'manager';
```

口诀：带头大哥不能死，中间兄弟不能断。

2）索引列上不计算

```
# 现在要查询`name` = 'Ringo'的记录下面有两种方式来查询!

# 1、直接使用 字段 = 值的方式来计算
mysql> SELECT * FROM `staffs` WHERE `name` = 'Ringo';
+----+-------+-----+---------+---------------------+
| id | name  | age | pos     | add_time            |
+----+-------+-----+---------+---------------------+
|  1 | Ringo |  18 | manager | 2020-08-03 08:30:39 |
+----+-------+-----+---------+---------------------+
1 row in set (0.00 sec)

# 2、使用MySQL内置的函数
mysql> SELECT * FROM `staffs` WHERE LEFT(`name`, 5) = 'Ringo';
+----+-------+-----+---------+---------------------+
| id | name  | age | pos     | add_time            |
+----+-------+-----+---------+---------------------+
|  1 | Ringo |  18 | manager | 2020-08-03 08:30:39 |
+----+-------+-----+---------+---------------------+
1 row in set (0.00 sec)
```

我们发现以上两条SQL的执行结果都是一样的，但是执行效率有没有差距呢？？？

通过分析两条SQL的执行计划来分析性能。

```
mysql> EXPLAIN SELECT * FROM `staffs` WHERE `name` = 'Ringo';
+----+-------------+--------+------------+------+-----------------------+-----------------------+---------+-------+------+----------+-------+
| id | select_type | table  | partitions | type | possible_keys         | key                   | key_len | ref   | rows | filtered | Extra |
+----+-------------+--------+------------+------+-----------------------+-----------------------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | staffs | NULL       | ref  | idx_staffs_name_age_pos | idx_staffs_name_age_pos | 74      | const |    1 |   100.00 | NULL  |
+----+-------------+--------+------------+------+-----------------------+-----------------------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT * FROM `staffs` WHERE LEFT(`name`, 5) = 'Ringo';
+----+-------------+--------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table  | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+--------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | staffs | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    3 |   100.00 | Using where |
+----+-------------+--------+------------+------+---------------+------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

索引没有用上

由此可见，在索引列上进行计算，会使索引失效。

口诀：索引列上不计算。

3）范围之后全失效

```sql
# 用到了idx_staffs_name_age_pos索引中的name，age，pos字段 这是属于全值匹配的情况！！！
EXPLAIN SELECT * FROM `staffs` WHERE `name` = 'Ringo' AND `age` = 18 AND `pos` = 'manager';


# 用到了idx_staffs_name_age_pos索引中的name，age字段，pos字段索引失效
EXPLAIN SELECT * FROM `staffs` WHERE `name` = '张三' AND `age` > 18 AND `pos` = 'dev';

# 解决：在业务允许的情况下，加上“=”
EXPLAIN SELECT * FROM `staffs` WHERE `name` = '张三' AND `age` >= 18 AND `pos` = 'dev';
```

查看上述SQL的执行计划

```
mysql> EXPLAIN SELECT * FROM `staffs` WHERE `name` = 'Ringo' AND `age` = 18 AND `pos` = 'manager';
+----+-------------+--------+------------+------+-----------------------+-----------------------+---------+-------------------+------+----------+-------+
| id | select_type | table  | partitions | type | possible_keys         | key                   | key_len | ref               | rows | filtered | Extra |
+----+-------------+--------+------------+------+-----------------------+-----------------------+---------+-------------------+------+----------+-------+
|  1 | SIMPLE      | staffs | NULL       | ref  | idx_staffs_name_age_pos | idx_staffs_name_age_pos | 140     | const,const,const |    1 |   100.00 | NULL  |
+----+-------------+--------+------------+------+-----------------------+-----------------------+---------+-------------------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT * FROM `staffs` WHERE `name` = 'zs' AND `age` > 18 AND `pos` = 'dev';
+----+-------------+--------+------------+-------+-----------------------+-----------------------+---------+------+------+----------+-----------------------+
| id | select_type | table  | partitions | type  | possible_keys         | key                   | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+--------+------------+-------+-----------------------+-----------------------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | staffs | NULL       | range | idx_staffs_name_age_pos | idx_staffs_name_age_pos | 78      | NULL |    1 |    33.33 | Using index condition |
+----+-------------+--------+------------+-------+-----------------------+-----------------------+---------+------+------+----------+-----------------------+
1 row in set, 1 warning (0.00 sec)
```

name age pos 三个字段都用到了

pos索引字段失效

由此可知，查询范围的字段使用到了索引，但是范围之后的索引字段会失效。

口诀：范围之后全失效。

4）覆盖索引尽量用

在写SQL的不要使用 `SELECT *` ，用什么字段就查询什么字段，且这些字段应该是索引字段，如果不是索引字段，则很有可能出现回表查询。

在Extra字段中，

- using index condition： 查询使用了索引，但是需要回表查询数据
- Using where;using index：查找使用了索引，但是需要的数据都在索引列中能找到，不需要回表查询数据
- 当然版本不同，出现的结果可能会不一样

```
# 聚集索引，叶子节点直接返回整行数据
EXPLAIN SELECT * FROM `staffs` WHERE id=20;

# 给name字段，创建索引
create index idx_staffs_name on staffs(name)

# 使用了覆盖索引 -- Extra:Using index
explain select id,name from staffs where name='Rose'

# 没有使用覆盖索引 -- Extra:NULL
# 其中，phone字段没有定义索引
explain select id,name,phone from staffs where name='Rose';

# 以下也没有使用覆盖索引，*包含了其它非索引字段 -- 必定会进行回表查询，从而影响性能
explain select * from staffs where name='Rose';
```

口诀：查询一定不用 `*` 。

5）like百分加右边

通过符%不能写在开始位置，否则索引失效。

```
# 索引失效 全表扫描
EXPLAIN SELECT * FROM `staffs` WHERE `name` LIKE '%ing%';

# 索引失效 全表扫描
EXPLAIN SELECT * FROM `staffs` WHERE `name` LIKE '%ing';

# 使用索引范围查询
EXPLAIN SELECT * FROM `staffs` WHERE `name` LIKE 'Rin%';
```

口诀： `like` 百分加右边。

如果一定要使用 `%like` ，而且还要保证索引不失效，那么使用覆盖索引来编写SQL。

```
# 使用到了覆盖索引
EXPLAIN SELECT `id` FROM `staffs` WHERE `name` LIKE '%in%';
```

```
# 使用到了覆盖索引
EXPLAIN SELECT `name` FROM `staffs` WHERE `name` LIKE '%in%';

# 使用到了覆盖索引
EXPLAIN SELECT `age` FROM `staffs` WHERE `name` LIKE '%in%';

# 使用到了覆盖索引
EXPLAIN SELECT `pos` FROM `staffs` WHERE `name` LIKE '%in%';

# 使用到了覆盖索引
EXPLAIN SELECT `id`, `name` FROM `staffs` WHERE `name` LIKE '%in%';

# 使用到了覆盖索引
EXPLAIN SELECT `id`, `age` FROM `staffs` WHERE `name` LIKE '%in%';

# 使用到了覆盖索引
EXPLAIN SELECT `id`,`name`, `age`, `pos` FROM `staffs` WHERE `name` LIKE '%in';

# 使用到了覆盖索引
EXPLAIN SELECT `id`, `name` FROM `staffs` WHERE `pos` LIKE '%na';

# 索引失效 全表扫描 , 覆盖索引 不包含 add_time
EXPLAIN SELECT `name`, `age`, `pos`, `add_time` FROM `staffs` WHERE `name` LIKE '%in';
```

```
mysql> EXPLAIN SELECT `name`, `age`, `pos` FROM `staffs` WHERE `name` LIKE '%in%';
+----+-------------+--------+------------+-------+---------------+-------------------------+---------+------+------+----------+--------------------------+
| id | select_type | table  | partitions | type  | possible_keys | key                     | key_len | ref  | rows | filtered | Extra                    |
+----+-------------+--------+------------+-------+---------------+-------------------------+---------+------+------+----------+--------------------------+
|  1 | SIMPLE      | staffs | NULL       | index | NULL          | idx_staffs_name_age_pos | 140     | NULL |    3 |    33.33 | Using where; Using index |
+----+-------------+--------+------------+-------+---------------+-------------------------+---------+------+------+----------+--------------------------+
1 row in set, 1 warning (0.00 sec)
```

口诀：覆盖索引保两边。

6）字符要加单引号

字段串类型的索引字段，在使用时必须加引号，否则索引失效

```
# 符合最左法则，部分使用了索引
EXPLAIN SELECT * FROM `staffs` WHERE `name` = '2000';

# 字符串类型字段不加引号，索引失效
# 这里name = 2000在MySQL中会发生强制类型转换，将数字转成字符串。
EXPLAIN SELECT * FROM `staffs` WHERE `name` = 2000;

# 以下查询是否走索引
EXPLAIN SELECT * FROM `staffs` WHERE `name` = 'Ringo' AND `age` = 18 AND `pos` = 1000;
```

口诀：字符要加单引号。

7）or连接的条件

使用or分割的条件中，如果or前的条件中字段有索引，而后面的字段中没有索引，那么涉及的索引都不会被用到。

```
# 其中，id字段有主键索引，add_time字段没有定义索引
# 解决：给add_time字段添加索引
EXPLAIN SELECT * FROM `staffs` WHERE id=1 or add_time='2021-03-01';

# 索引失效，全表查询，or的右边也必须遵循最左原则
EXPLAIN SELECT * FROM `staffs` WHERE id=1 or age=20;

# 索引生效
EXPLAIN SELECT * FROM `staffs` WHERE id=1 or name='李四';
```

8）数据分布影响

如果MySQL评估使用索引比值班表更慢，则不使用索引。

```
# 全表查询
explain SELECT * from staffs where phone>='13417740000';


# 给phone字段添加索引
create index idx_staffs_phone on staffs(phone)

# 再次测试，发现仍然是全表查询，并没有使用到索引
# 原因：当满足条件时，相当于查询全表，MySQL认为查询全表比按索引查询更快
explain SELECT * from staffs where phone>='13417740000';

# 修改条件再测试，满足条件只是部分数据，进索引更快
explain SELECT * from staffs where phone>='13417740009';


# 思考，执行以下SQL语句，是否走索引?
EXPLAIN SELECT * FROM `staffs` WHERE `phone` != '13417740001';

# 删除数据，再测试
DELETE from staffs where phone>='13417740004'
```

9）SQL提示

SQL提示，是优化数据库的一个重要手段。简单的说，就是在SQL语句中加入一些人为的来达到优化操作的目的。

- use index：使用某个索引，但MySQL不一定使用
- Ignore index：忽略某个索引
- force index：强制使用某个索引

```
# 语法
select 字段集合 from 表名 use index(索引名称) where 条件 ;
select 字段集合 from 表名 ignore index(索引名称) where 条件 ;
select 字段集合 from 表名 force index(索引名称) where 条件 ;


# 创建单列索引
CREATE INDEX idx_staffs_name ON `staffs`(`name`);

# 测试，可能用到的索引有：idx_staffs_name_age_pos、idx_staffs_name
# 最终，MySQL选择了idx_staffs_name_age_pos
# 如果我们想使用idx_staffs_name索引，怎么办呢?
EXPLAIN SELECT * FROM `staffs` WHERE name='lucy';

# 使用指定的索引
EXPLAIN SELECT * FROM `staffs` use index(idx_staffs_name) WHERE name='lucy';

# 忽略某个索引
EXPLAIN SELECT * FROM `staffs` ignore index(idx_staffs_name_age_pos) WHERE name='lucy';

# 强制使用某个索引
EXPLAIN SELECT * FROM `staffs` force index(idx_staffs_name) WHERE name='lucy';
```

10）前缀索引

当字段类型为字符串时，字符串的长度可能很大，创建索引时，如果都使用全部字符串来创建，则索引会很大，查询时，浪费大量的IO，从而影响查询效率。因此，我们可以截取字符串的一小部分作为前缀，用于创建索引，这样可以大大节约索引空间，从而提高索引效率。

```
CREATE [UNIQUE] INDEX 索引名称 ON 表名 (字段(长度)) ;
```

如何设置字段前缀的长度

- 可以根据索引的选择性来决定；
- **选择性**是指不重复的索引值（基数）和数据表的记录总数的比值，索引选择性越高则查询效率越高；
- 唯一索引的选择性是1，这是最好的索引选择性，性能也是最好的。

```
# 查看某个字段的 选择性
select count(distinct(字段)) / count(*) from 表

# 查看某个字段前缀为N的 选择性
select count(distinct(left(字段,N))) / count(*) from 表

# 查看截取前面两个字符的选择性
select count(distinct(left(name,2))) / count(*) from staffs;
```