一、what-是什么

1、介绍

1.1 索引定义

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。

一句话：索引是数据结构。严格说，是一种排好序的数据结构。

1.2 常见的索引类型

主键索引（PRIMARY）、唯一索引（UNIQUE）、普通索引（INDEX）、组合索引、全文索引

最左前缀”原则：最常用作为检索或排序的列放在最左，依次递减

2、前世今生

2.1 索引目的

一句话：提高查询效率。

可以类比字典，如果要查“mysql”这个单词，我们肯定需要定位到m字母，然后从下往下找到y字母，再找到剩下的sql。如果没有索引，单词呢？如果没有索引，这个事情根本无法完成？

索引例子除了字典，还有图书目录。它们的原理都是一样的，通过不断的缩小想要获得数据的范围来筛选出最终想要的结果，同时把随

2.2 索引数据结构

数据库也是一样，但显然要复杂许多，因为不仅面临着等值查询，还有范围查询(>、<、between、in)、模糊查询(like)、并集查询(or)等段。

每一次IO读取的数据我们称之为一页(page)。具体一页有多大数据跟操作系统有关，一般为4k或8k，也就是我们读取一页内的数据时候为了减少磁盘IO，磁盘往往会进行数据预读，会从某位置开始，预先向后读取一定长度的数据放入内存，即局部性原理。因为磁盘顺序

综上分析，索引的数据结构，需要满足以下条件：每次查找数据时把磁盘IO次数控制在一个很小的数量级，最好是常数数量级。b+树应

2.3 索引分类

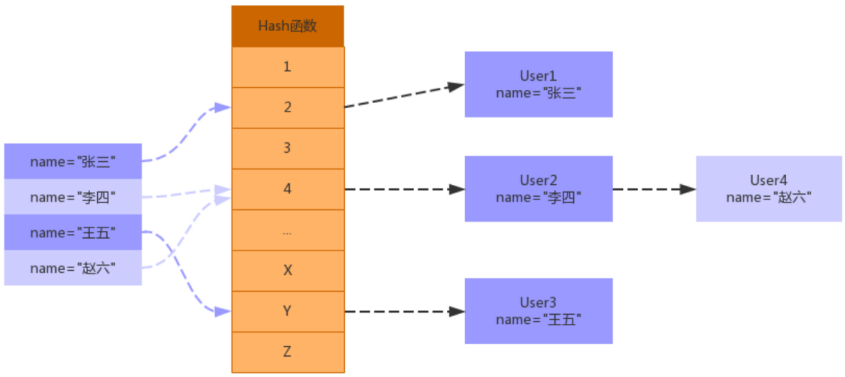
在数据库中，索引是分类，不只有B+树，还有Hash。B+树后面重点介绍，这里先说下Hash索引。

Hash结构由Hash表来实现的，是根据键值 <key,value> 存储数据的结构；Hash索引可以方便的提供等值查询，对于范围查询就需要全自适应哈希索引。

Hash 索引的单条记录查询的效率很高，时间复杂度为1。但是，Hash索引并不常用，主要有以下原因：

- Hash索引适合精确查找，但是范围查找不适合

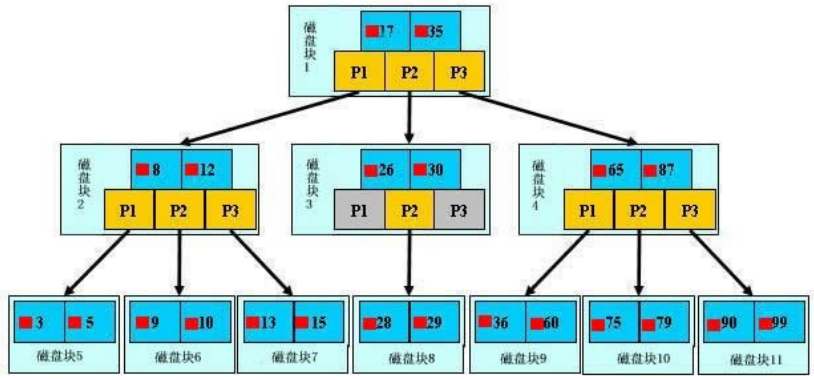
因为存储引擎都会为每一行计算一个hash码，hash码都是比较小的，并且不同键值行的hash码通常是不一样的，hash索引中存储的就是所以值相近的两个数据，Hash值相差很远，被分到不同的桶中。这就是为什么hash索引只能进行全职匹配的查询，因为只有这样，hash码



二、why-索引原理

1、详解B+树

B+树是B树的加强版，即Plus版。



浅蓝色的块我们称之为一个磁盘块，可以看到每个磁盘块包含几个数据项（深蓝色所示）和指针（黄色所示）。特点：

- 所有的关键字全部存储在叶子节点上，且叶子节点本身根据关键字自小而大顺序连接。
- 非叶子节点可以看成索引部分，节点中仅含有其子树（根节点）中的最大（或最小）关键字。
- B+Tree中的非叶子结点不存储数据，只存储键值。
- B+Tree的每个非叶子节点由n个键值key和n个指针point组成；每个叶子节点增加一个指向相邻叶子节点的指针，就形成了带有顺序访

1.1 b+树的查找过程

如图所示，如果要查找数据项29，步骤如下：

1. 首先，会把磁盘块1由磁盘加载到内存，此时发生一次IO，在内存中用二分查找确定29在17和35之间，锁定磁盘块1的P2指针，内存时
2. 其次，通过磁盘块1的P2指针的磁盘地址把磁盘块3由磁盘加载到内存，发生第二次IO，29在26和30之间，锁定磁盘块3的P2指针。
3. 然后，通过指针加载磁盘块8到内存，发生第三次IO，同时内存中做二分查找找到29，结束查询，总计三次IO。
4. 真实的情况是，3层的b+树可以表示上百万的数据，如果上百万的数据查找只需要三次IO，性能提高将是巨大的，如果没有索引，每

1.2 b+树性质

通过上面的分析，我们知道IO次数取决于b+数的高度h。

假设当前数据表的数据为N，每个磁盘块的数据项的数量是m，则有h=log(m+1)N，当数据量N一定的情况下，m越大，h越小。

而m = 磁盘块的大小 / 数据项的大小，磁盘块的大小也就是一个数据页的大小，是固定的，如果数据项占的空间越小，数据项的数量越

这就是为什么每个数据项，即索引字段要尽量的小，比如int占4字节，要比bigint8字节少一半。

这也是为什么b+树要求把真实的数据放到叶子节点而不是内层节点，一旦放到内层节点，磁盘块的数据项会大幅度下降，导致树增高。

1.3 B+组合索引结构

当b+树的数据项是复合的数据结构，比如(name,age,sex)的时候，b+数是按照从左到右的顺序来建立搜索树的。

比如当(张三,20,F)这样的数据来检索的时候，b+树会优先比较name来确定下一步的所搜方向，如果name相同再依次比较age和sex，最
道下一步该查哪个节点，因为建立搜索树的时候name就是第一个比较因子，必须要先根据name来搜索才能知道下一步去哪里查询。

比如当(张三,F)这样的数据来检索时，b+树可以用name来指定搜索方向，但下一个字段age的缺失，所以只能把名字等于张三的数据都

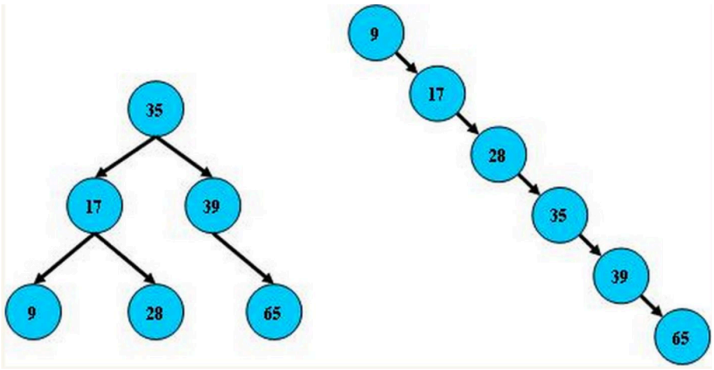
这个是非常重要的性质，即索引的最左匹配特性。

2、二叉树、平衡二叉树、B树对比

2.1 二叉排序树（Binary Sort Tree）

树上的一根树枝开两个叉，于是递归下来就是二叉树了，而这棵树上的节点是已经排好序的，具体的排序规则如下：

- 若左子树不空，则左子树上所有节点的值均小于它的根节点的值
- 若右子树不空，则右字数上所有节点的值均大于它的根节点的值
- 它的左、右子树也分别为二叉排序数（递归定义）



优势：提前排好序了，查找比较方便、比较快。

劣势：极端情况会出现所有节点都位于同一侧，直观上看就是一条直线。

2.2 平衡二叉树

平衡二叉树，就是对二叉树的优化，避免极端情况出现。

为了保证树的结构左右两端数据大致平衡，降低二叉树的查询难度，一般会采用一种算法机制实现节点数据结构的平衡，实现了这种算

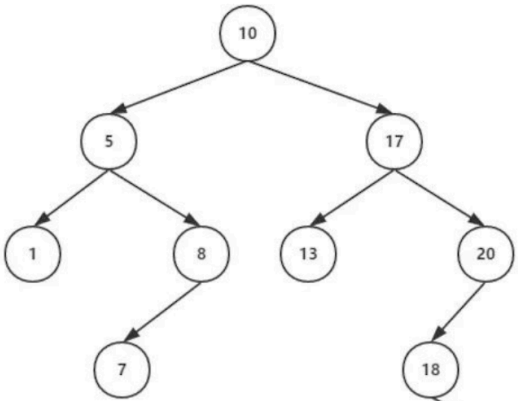
所谓“平衡”，说的是这棵树的各个分支的高度是均匀的，它的左子树和右子树的高度之差绝对值小于1，这样就不会出现一条支路特别长的高度，这就确保了查询的效率（时间复杂度为O(logn)）。总结平衡二叉树特点：

（1）非叶子节点最多拥有两个子节点；

（2）非叶子节值大于左边子节点、小于右边子节点；

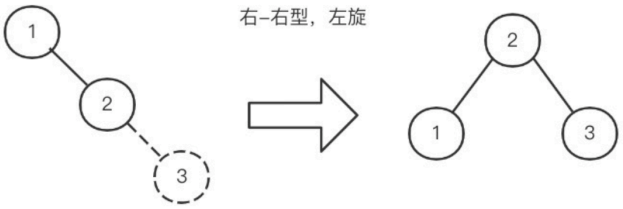
（3）树的左右两边的层级数相差不会大于1;

（4）没有值相等重复的节点;



出现二叉树的问题时怎么解决呢?

那应该怎么办呢?因为它是右节点下面接一个右节点，右-右型，所以这个时候我们要把 2 提上去，这个操作叫做左旋。



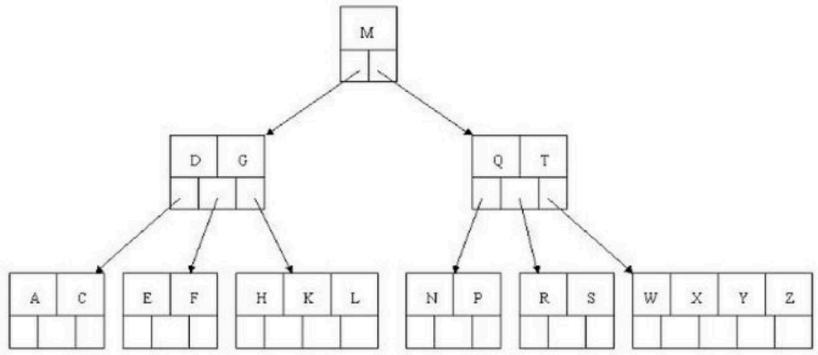
同样的，如果我们插入3、2、1，这个时候会变成左左型，就会发生右旋操作，把 2 提上去。

所以为了保持平衡，AVL 树在插入和更新数据的时候执行了一系列的计算和调整的操作。

2.3 B树

B树属于多叉树，又名平衡多路查找树（查找路径不只两个）。规则如下：

- 排序方式：所有节点关键字是按递增次序排列，并遵循左小右大原则；
- 子节点数：非叶节点的子节点数>1，且<=M ，且M>=2，空树除外（注：M阶代表一个树节点最多有多少个查找路径，M=M路,当M=
- 关键字数：枝节点的关键字数量大于等于ceil(m/2)-1个且小于等于M-1个（注：ceil()是个朝正无穷方向取整的函数 如ceil(1.1)结果为2
- 所有叶子节点均在同一层、叶子节点除了包含了关键字和关键字记录的指针外也有指向其子节点的指针只不过其指针地址都为null对应

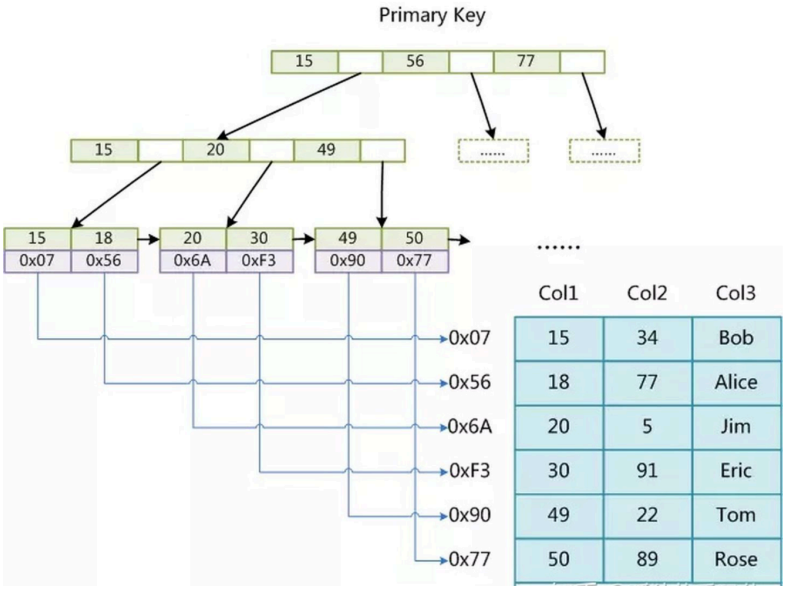


3、MySQL索引实现

MySQL中常见的有MyISAM和InnoDB两个存储引擎的索引实现方式。

3.1 MyISAM

MyISAM引擎使用B+Tree作为索引结构，叶节点的data域存放的是数据记录的地址。原理图：



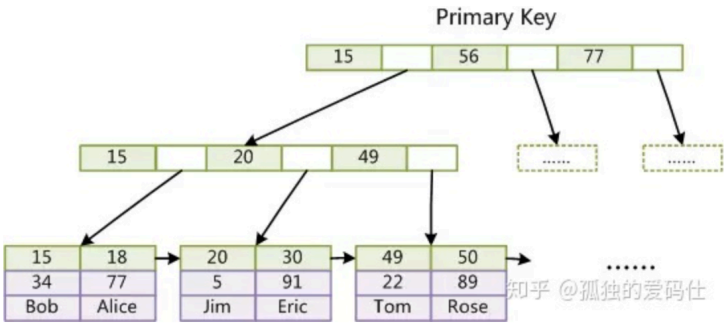
MyISAM的索引文件仅仅保存数据记录的地址。在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索

3.2 InnoDB

InnoDB也使用B+Tree作为索引结构，但具体实现方式却与MyISAM截然不同。区别如下：

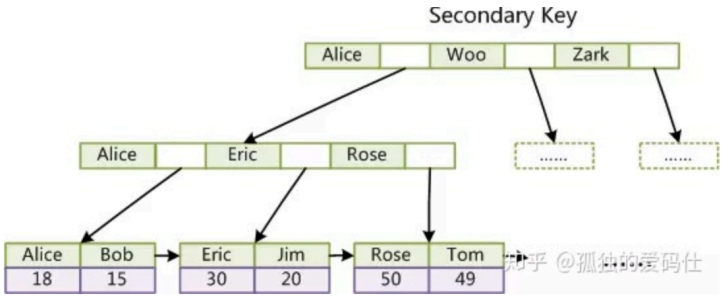
- **InnoDB的数据文件本身就是索引文件**。

MyISAM索引文件和数据文件是分离的，索引文件仅保存数据记录的地址。而在InnoDB中，表数据文件本身就是按B+Tree组织的一个索表的主键，因此InnoDB表数据文件本身就是主索引。



- **InnoDB的辅助索引data域存储相应记录主键的值而不是地址**

也就是说，InnoDB的所有辅助索引都引用主键作为data域。



聚集索引这种实现方式使得按主键的搜索十分高效，但是辅助索引搜索需要检索两遍索引：首先检索辅助索引获得主键，然后用主键到

知道了InnoDB的索引实现后，就明白为什么不建议使用过长的字段作为主键，因为所有辅助索引都引用主索引，过长的主索引会令辅助

三、how-如何使用

1、创建索引

1.1 创建表时加索引

```
CREATE TABLE mytable(
    id INT NOT NULL,
    name VARCHAR(16) NOT NULL,
    INDEX [idx_name] (name(length)))
);
```

1.2 创建表后加索引

```
ALTER TABLE my_table ADD [UNIQUE] INDEX index_name(column_name);
或者
CREATE INDEX index_name ON my_table(column_name);
```

2、删除索引

```
DROP INDEX my_index ON tablename;
或者
ALTER TABLE table_name DROP INDEX index_name;
```

3、建索引的几大原则

1）最左前缀匹配原则

mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配。

比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,

2）=和in可以乱序

比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式。

3）尽量选择区分度高的列作为索引

区分度的公式是count(distinct col)/count(*)，表示字段不重复的比例，比例越大我们扫描的记录数越少，唯一键的区分度是1，而一些

这个比例有什么经验值吗？使用场景不同，这个值也很难确定，一般需要join的字段我们都要求是0.1以上，即平均1条扫描10条记录。

4）索引列不能参与计算，保持列“干净”

比如from_unixtime(create_time) = ’2014-05-29’就不能使用到索引。

原因很简单，b+树中存的都是数据表中的字段值，但进行检索时，需要把所有元素都应用函数才能比较，显然成本太大。所以语句应该

5）尽量的扩展索引，不要新建索引

比如表中已经有a的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可。

4、SQL优化

4.1 explain

查看一个sql有没有用的索引，以及索引情况。

举例：

```
EXPLAIN SELECT * from user WHERE id < 3;
```

执行结果：

```
mysql> explain select * from user where id<3 \G;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: user
   partitions: NULL
         type: range
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: NULL
         rows: 2
     filtered: 100.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)
```

结果说明：

1）select_type

select_type表示查询的类型。常见的值如下：

- SIMPLE： 表示查询语句不包含子查询或union；

- PRIMARY：表示此查询是最外层的查询；
- UNION：表示此查询是UNION的第二个或后续的查询；
- DEPENDENT UNION：UNION中的第二个或后续的查询语句，使用了外面查询结果；
- UNION RESULT：UNION的结果；
- SUBQUERY：SELECT子查询语句；
- DEPENDENT SUBQUERY：SELECT子查询语句依赖外层查询的结果。

2） type

type表示存储引擎查询数据采用的方式。可以判断出查询是全表扫描还是基于索引的部分扫描。常用属性值如下，从上至下效率依次增强：

- ALL：表示全表扫描，性能最差。
- index：表示基于索引的全表扫描，先扫描索引再扫描全表数据。
- range：表示使用索引范围查询。使用>、>=、<、<=、in等等。
- ref：表示使用非唯一索引进行单值查询。
- eq_ref：一般情况下出现在多表join查询，表示前面表的每一个记录，都只能匹配后面表的一行结果。
- const：表示使用主键或唯一索引做等值查询，常量查询。
- NULL：表示不用访问表，速度最快。

3） Extra

Extra表示很多额外的信息。各种操作会在Extra提示相关信息，常见几种如下：

- Using where：表示查询需要通过索引回表查询数据。
- Using index：表示查询需要通过索引，索引就可以满足所需数据。
- Using filesort：表示查询出来的结果需要额外排序，数据量小在内存，大的话在磁盘，因此有Using filesort建议优化。
- Using temprorary：查询使用到了临时表，一般出现于去重、分组等操作。

4） key

key表示查询时真正使用到的索引，显示的是索引名称。

5） rows

MySQL查询优化器会根据统计信息，估算SQL要查询到结果需要扫描多少行记录。

6） key_len

key_len 表示查询使用索引的字节数量。可以判断是否全部使用了组合索引，key_len的计算规则如下：

字符串类型：字符串长度跟字符集有关：latin1=1、gbk=2、utf8=3、utf8mb4=4

- char(n)：n*字符集长度
- varchar(n)：n * 字符集长度 + 2字节

数值类型：

- TINYINT：1个字节
- SMALLINT：2个字节
- MEDIUMINT：3个字节
- INT、FLOAT：4个字节
- BIGINT、DOUBLE：8个字节

时间类型：

- DATE：3个字节
- TIMESTAMP：4个字节
- DATETIME：8个字节

字段属性：

- NULL属性占用1个字节，如果一个字段设置了NOT NULL，则没有此项

7） possible_keys

possible_keys 表示查询时能够使用到的索引。注意并不一定会真正使用，显示的是索引名称。

4.2 慢查询

查询是否使用索引，只是表示一个SQL语句的执行过程；而是否为慢查询，是由它执行的时间决定的，也就是说是否使用了索引和是否在使用索引时，不要只关注是否起作用，应该关心索引是否减少了查询扫描的数据行数；对于一个大表，不止要创建索引，还要考虑索

1）慢查询日志

慢查询日志相关参数命令设置：

```
-- 查看数据库是否开启了慢查询日志：
  SHOW VARIABLES LIKE 'slow_query_log%';
-- 开启慢查询日志命令：
  SET global slow_query_log = ON;
-- 修改日志文件名称：
```

```
  SET global slow_query_log_file = 'fishleap-slow.log';
-- 记录没有使用索引的查询SQL：前提是slow_query_log的值为ON
  SET global log_queries_not_using_indexes = ON;
-- 指定慢查询的阀值，单位秒：
  SET long_query_time = 10;
```

2）查看慢查询日志

- 打开slow.log日志即可：

```
Time                 Id Command    Argument
# Time: 2020-03-10T14:18:02.145102Z
# User@Host: root[root] @ localhost [::1]  Id:     5
# Query_time: 1316.989328  Lock_time: 0.000000 Rows_sent: 0  Rows_examined: 0
use oak;
SET timestamp=1583849882;
select * from dept;
```

文件分析：

```
time：日志记录的时间
User@Host：执行的用户及主机
Query_time：执行的时间
Lock_time：锁表时间
Rows_sent：发送给请求方的记录数，结果数量
Rows_examined：语句扫描的记录条数
SET timestamp：语句执行的时间点
select....：执行的具体的SQL语句
```

3）慢查询优化基本步骤

1. 先运行看看是否真的很慢，注意设置SQL_NO_CACHE
2. where条件单表查，锁定最小返回记录表。把查询语句的where都应用到表中返回的记录数最小的表开始查起，单表每个字段分别查
3. explain查看执行计划，是否与1预期一致（从锁定记录较少的表开始查询）
4. order by limit 形式的sql语句让排序的表优先查
5. 了解业务方使用场景
6. 加索引时参照建索引的几大原则
7. 观察结果，不符合预期继续从0分析

举例：

```
-- SQL案例：
  select * from student where age=18 and name like '张%'; (问题：全表扫描)

-- 优化1：
  alter table student add index(name); //追加name索引
-- 优化2：
  alter table student add index(age,name); //追加age,name索引 (index condition pushdown 优化的效果)
-- 优化3：为user表添加first_name虚拟列，以及联合索引(first_name,age)
  alter table student add first_name varchar(2) generated always as  (left(name, 1)), add index(first_name,
```

5、高质量SQL建议

1）查询

- 查询SQL尽量使用select具体字段；
- 查询结果只有一条记录时建议使用 limit 1；
- 插入数据过多，考虑批量插入（foreach标签）；
- 字段过多时慎用 distinct 关键字；
- 删除冗余和重复索引；
- 删除/修改数据量过大建议分批进行删除；

2）where子句

- 避免where子句中使用 or 来连接条件（union all）；
- 优化like语句（%放关键字后面），若不走索引可以使用覆盖索引；
- 避免返回多余的行（条件唯一）；
- 避免在索引列上使用内置函数；

- 避免对字段进行表达式操作；
- 避免使用 != 或 <>（考虑分两条sql写）；
- 考虑在 where 及 order by 涉及的列上建立索引；
- 考虑使用默认值代替null；

3）limit分页

- 返回上次查询的最大记录（where id > 10000） limit 10；
- 表连接：
- 都满足SQL需求前提下，优先使用inner join，left join左边数据尽量小；

4）联合索引

- 遵循最左匹配原则；

参考文献：

1、【美团】MySQL索引原理及慢查询优化：https://tech.meituan.com/2014/06/30/mysql-index.html
2、【知乎】深入理解MySQL索引底层实现原理：https://zhuanlan.zhihu.com/p/77383599
3、MySQL索引分析与优化、慢查询优化：http://fishleap.top/pages/bb256a/#%E7%9F%A5%E8%AF%86%E7%82%B9