20240314腾讯研发管理一面

项目

八股

排序算法的稳定性

https://cloud.tencent.com/developer/article/1182321

排序算法的稳定性是指：排序前后两个相等的数相对位置不变，则算法稳定。

> 稳定的排序算法有以下几种：
>
> 1. 冒泡排序（Bubble Sort）
> 2. 归并排序（Merge Sort）
> 3. 插入排序（Insertion Sort）
> 4. 计数排序（Counting Sort）
> 5. 基数排序（Radix Sort）
>
> CSDN @阿年、嗯啊

桶排序可以是稳定的。桶排序的实现中，元素是通过将其分配到为每个桶预先定义的范围内来分类的，而每个桶之间是没有关系的，所以对于输入的序列中相同大小的元素，它们会被分配到同一个桶中，即在桶内部它们顺序不变，从而保证了相对位置的稳定性。因此，**如果在桶内使用稳定的排序算法，则可以实现稳定的桶排序**。然而，在对桶进行划分时，如何设置桶的数量和其划分范围会影响最终结果，需要根据具体需求进行调整。

冒泡排序

1. 小的元素往前调或者把大的元素往后调

2. 比较是相邻的两个元素比较，交换也发生在这两个元素之间，是**稳定排序算法**。

选择排序

1. 首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置
2. 再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。
3. 重复第 2 步，直到所有元素均排序完毕。

举个例子，序列5 8 5 2 9， 第一遍选择时最小的元素为2，第1个元素5会和2交换，那么原序列中2个5的相对前后顺序就被破坏了；**不稳定的排序算法**。

插入排序

1. 从第一个元素开始，该元素可以认为已经被排序；
2. 取出下一个元素，在已经排序的元素序列中从后向前扫描；
3. 如果该元素（已排序）大于新元素，将该元素移到下一位置；
4. 重复步骤 3，直到找到已排序的元素小于或者等于新元素的位置；
5. 将新元素插入到该位置后；
6. 重复步骤 2~5。

如果碰见一个和插入元素相等的，那么插入元素把想插入的元素放在相等元素的后面，相等元素的前后顺序没有改变；**稳定排序算法**。

快速排序

在中枢元素和a[j]交换的时候，很有可能把前面的元素的稳定性打乱，比如序列为 5 3 3 4 3 8 9 10 11， 现在中枢元素5和3(第5个元素，下标从1开始计)交换就会把元素3的稳定性打乱；不稳定发生在中枢元素和a[j] 交换的时刻；**不稳定的排序算法**。

归并排序

在合并过程中我们可以保证如果两个当前元素相等时，我们把处在前面的序列的元素保存在结果序列的前面，这样就保证了稳定性；**稳定排序算法**。

希尔排序

由于会进行多次插入排序，一次插入排序是稳定的，不会改变相同元素的相对顺序，但在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性就会被打乱；**不稳定的排序算法**。

基数排序

按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位；**稳定排序算法**。

堆排序

不稳定的排序算法。

mysql不同的存储引擎

1. 数据结构的区别

2. 聚簇索引、非聚簇索引的区别
3. 为什么要用B+树

慢查询怎么优化

https://tech.meituan.com/2014/06/30/mysql-index.html

https://zhuanlan.zhihu.com/p/149338523

https://cloud.tencent.com/developer/article/1545163

索引没起作用的情况

- `SELECT *` 不会直接导致索引失效（如果不走索引大概率是因为 where 查询范围过大导致的），但它可能会带来一些其他的性能问题比如造成网络传输和数据处理的浪费、无法使用索引覆盖;
- 创建了组合索引，但查询条件未遵守最左匹配原则;
- 在索引列上进行计算、函数、类型转换等操作;
- 以 % 开头的 LIKE 查询比如 `LIKE '%abc';`;
- 查询条件中使用 OR，且 OR 的前后条件中有一个列没有索引，涉及的索引都不会被使用到;
- IN 的取值范围较大时会导致索引失效，走全表扫描(NOT IN 和 IN 的失效场景相同);
- 发生隐式转换。

优化数据库结构

合理的数据库结构不仅可以使数据库占用更小的磁盘空间，而且能够使查询速度更快。数据库结构的设计，需要考虑数据冗余、查询和更新的速度、字段的数据类型是否合理等多方面的内容。

- 将字段很多的表分解成多个表。对于字段比较多的表，如果有些字段的使用频率很低，可以将这些字段分离出来形成新表。因为当一个表的数据量很大时，会由于使用频率低的字段的存在而变慢。
- 增加中间表。对于需要经常联合查询的表，可以建立中间表以提高查询效率。通过建立中间表，把需要经常联合查询的数据插入到中间表中，然后将原来的联合查询改为对中间表的查询，以此来提高查询效率。

分解关联查询

将一个大的查询分解为多个小查询是很有必要的。

很多高性能的应用都会对关联查询进行分解，就是可以对每一个表进行一次单表查询，然后将查询结果在应用程序中进行关联，很多场景下这样会更高效，例如：

```
SELECT * FROM tag
       JOIN tag_post ON tag_id = tag.id
       JOIN post ON tag_post.post_id = post.id
       WHERE tag.tag = 'mysql';

       分解为：
```

```
SELECT * FROM tag WHERE tag = 'mysql';
SELECT * FROM tag_post WHERE tag_id = 1234;
SELECT * FROM post WHERE post.id in (123,456,567);
```

优化LIMIT分页

在系统中需要分页的操作通常会使用limit加上偏移量的方法实现，同时加上合适的order by 子句。如果有对应的索引，通常效率会不错，否则MySQL需要做大量的文件排序操作。

一个非常令人头疼问题就是当偏移量非常大的时候，例如可能是limit 10000,20这样的查询，这是mysql需要查询10020条然后只返回最后20条，前面的10000条记录都将被舍弃，这样的代价很高。

方法一：先查询出主键id值

对于下面的查询：

select id,title from collect limit 90000,10;

该语句存在的最大问题在于limit M,N中偏移量M太大（我们暂不考虑筛选字段上要不要添加索引的影响），导致每次查询都要先从整个表中找到满足条件 的前M条记录，之后舍弃这M条记录并从第M+1条记录开始再依次找到N条满足条件的记录。如果表非常大，且筛选字段没有合适的索引，且M特别大那么这样的代价是非常高的。 试想，如我们下一次的查询能从前一次查询结束后标记的位置开始查找，找到满足条件的100条记录，并记下下一次查询应该开始的位置，以便于下一次查询能直接从该位置 开始，这样就不必每次查询都先从整个表中先找到满足条件的前M条记录，舍弃，在从M+1开始再找到100条满足条件的记录了。

```
select id,title from collect where id>=(select id from collec
t order by id limit 90000,1) limit 10;
```

原理：先查询出90000条数据对应的主键id的值，然后直接通过该id的值直接查询该id后面的数据。

方法二：用延迟关联优化分页（LIMIT）

https://juejin.cn/post/6953915051006754852

当使用 `LIMIT` 碰上较大偏移量时，例如 `LIMIT 10000, 20` 这样的查询，MySQL 需要查询 10020 条记录然后再返回最后的 20 条。前面的 10000 最终都会被抛弃，这样的代价非常高。

```
mysql> EXPLAIN SELECT * FROM orders LIMIT 10000, 20;
```

优化此类分页查询的一个最简单的办法就是**尽可能使用索引覆盖扫描（覆盖索引），而不是查询所有列**。然后根据需要再做一次关联，返回所需要的列。（即先在要查询的字段上建立索引，使用索引查询到目标的数据时，再回表做一次关联，返回完整的数据）

```
mysql> EXPLAIN SELECT * FROM orders AS o1 JOIN (SELECT id FRO
```

```
M orders LIMIT 10000, 20) AS o2 ON o1.id = o2.id;
```

这样一来，MySQL 在 SQL 语句的「内层」进行扫描时使用了覆盖索引，「外层」再通过索引树找到相关的数据行，直接减少了扫描的数据量。

方法三：建立联合索引

数据结构

1. hashmap
2. 栈和队列的应用

Https的加密

TCP三次握手

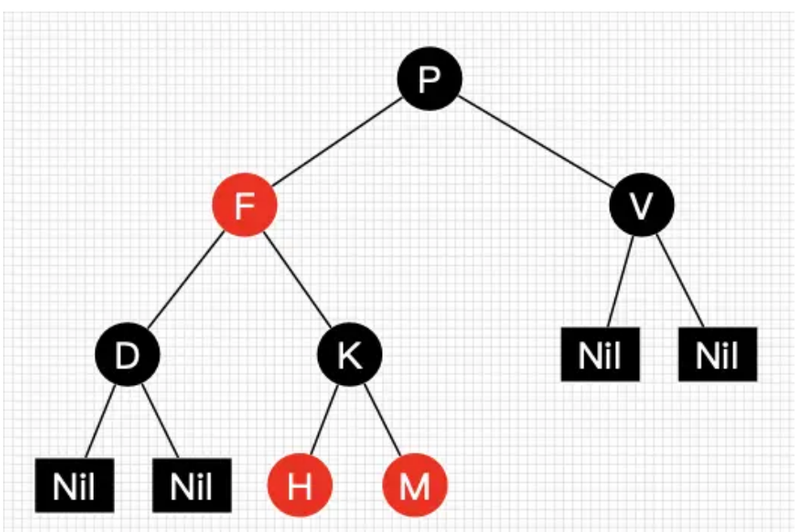
常用Linux命令

红黑树

https://www.jianshu.com/p/e136ec79235c

红黑树是一种含有红黑结点并能自平衡的二叉查找树。特点：

- 每个节点非红即黑；
- 根节点总是黑色的；
- 每个叶子节点都是黑色的空节点（NIL 节点）；
- 如果节点是红色的，则它的子节点必须是黑色的（反之不一定）；
- 从根节点到叶节点或空子节点的每条路径，必须包含相同数目的黑色节点（即相同的黑色高度）。



该图就是一颗简单的红黑树。其中Nil为叶子结点(**图中的红色结点H和M同样存在叶子子结点NIL**)，并且它是黑色的。

红黑树能自平衡，它靠的是三种操作：左旋、右旋和变色。**红黑树总是通过旋转和变色达到自平衡**。

- **左旋**：以某个结点作为支点(旋转结点)，其右子结点变为旋转结点的父结点，右子结点的左子结点变为旋转结点的右子结点，左子结点保持不变。
- **右旋**：以某个结点作为支点(旋转结点)，其左子结点变为旋转结点的父结点，左子结点的右子结点变为旋转结点的左子结点，右子结点保持不变。
- **变色**：结点的颜色由红变黑或由黑变红。

算法

1. 括号有效性
2. 原地修改数组