10 为什么添加索引能提高查询速度?

更新时间：2019-07-30 10:13:42

> 天才免不了有障碍，因为障碍会创造天才。
>
> ——罗曼·罗兰

从本节开始，专栏将进入一个新的 MySQL 知识大类：MySQL 索引。

在开始讲解本章内容之前，我们来简单回顾一下第一章（SQL 优化）的内容。我们讲解了分析 SQL 执行效率的几种方法，分享了几种字段有索引但是不走索引的情况，并讲解了几类常用的 SQL 的优化技巧，比如：数据导入、排序、分组、分页查询、关联查询、count() 等。

在第一章的学习中，可能你已经发现有好几个地方的优化是添加索引，那么索引为什么能提高查询速度呢？到底应该选择哪种类型的索引呢？我们可以带着这些疑问进入第二章（MySQL 索引）的学习。

比如一本比较厚的书，我们需要找到对应的知识点，我们的习惯一般都是先看目录，根据目录去找到对应的知识点。试想一下，假如这本比较厚的书没有目录，我们就需要从前面到后面一页一页地找，直到找到对应的知识点，这个过程估计得耗费一段时间了。

跟书本创建目录一样，我们使用 MySQL 时，会考虑在需要做条件查询的字段上添加索引。那么为什么添加索引能提高 MySQL 的查询速度呢？这一节就一起来聊聊这个话题吧。

为了便于理解 MySQL 的索引，我们先了解一些与索引相关的算法。

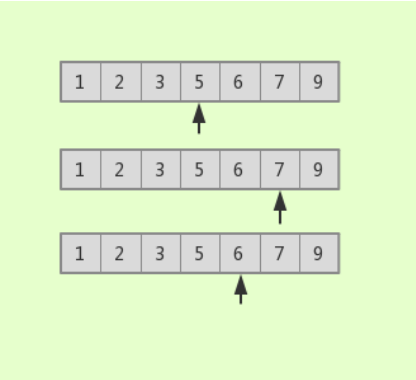
1 跟索引相关的一些算法

对于 MySQL 而言，使用最频繁的就是 B+ 树索引，所以我们必须要知道 B+ 树的结构，而 B+ 树是借鉴了二分查找法、二叉查找树、平衡二叉树、B 树的一些思想构建的。因此我们首先通过了解这些算法，来一层一层拨开 B+ 树的神秘面纱。

1.1 二分查找法

二分查找法的查找过程是：将记录按顺序排列，查找时先以有序列的中点位置为比较对象，如果要找的元素值小于该中点元素，则将查询范围缩小为左半部分；如果要找的元素值大于该中点元素，则将查询范围缩小为右半部分。以此类推，直到查到需要的值。

比如要从 1、2、3、5、6、7、9 几个数字中找到 6，首先找到中点位置的值，这里是 5，因为 6 大于 5，所以查询以 5 为中点右边的数字（其实就是大于 5 的数字），又因为 6 小于 7，所以继续查询以 7 为中点左边的数字，发现数字 6，返回结果。具体过程如下图：



发现用了 3 次就查找到 6 这个数字了。如果是顺序查找，则需要查询 5 次（从第一个数字 1 开始，如果发现不是 6，则继续查找下一个，直到查询到 6）。

我们来对比一下这个例子顺序查找和二分查找法的平均查找次数：

顺序查找：(1 + 2 + 3 + 4 + 5 + 6 + 7)/7 = 4 次

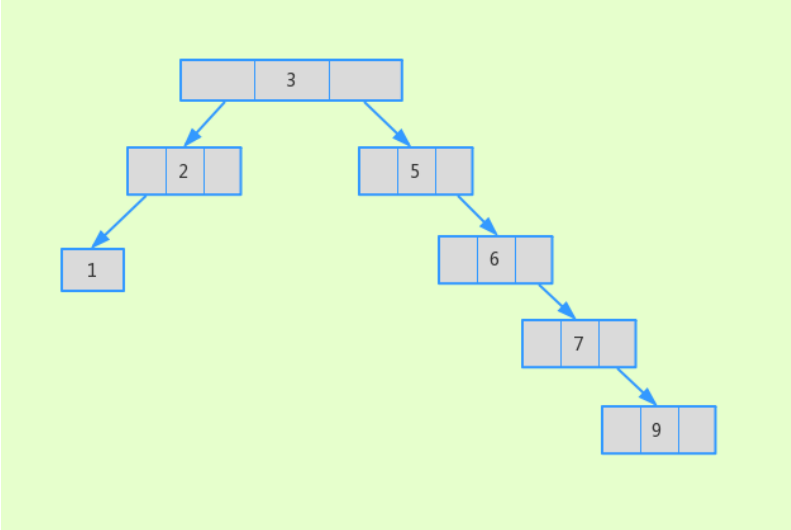
二分查找法：(3 + 2 + 3 + 1 + 3 + 2 + 3)/7 ≈ 2.4 次

显然二分查找法相对顺序查找平均效率更高。

1.2 二叉查找树

二叉查找树中，左子树的键值总是小于根的键值，右子树的键值总是大于根的键值，并且每个节点最多只有两颗子树。

对于 1.1 中举例的这组数字（1、2、3、5、6、7、9），其结构大致如下：



这组数字的平均查找次数为：(3 + 2 + 1 + 2 + 3 + 4 + 5)/7 ≈ 2.9 次

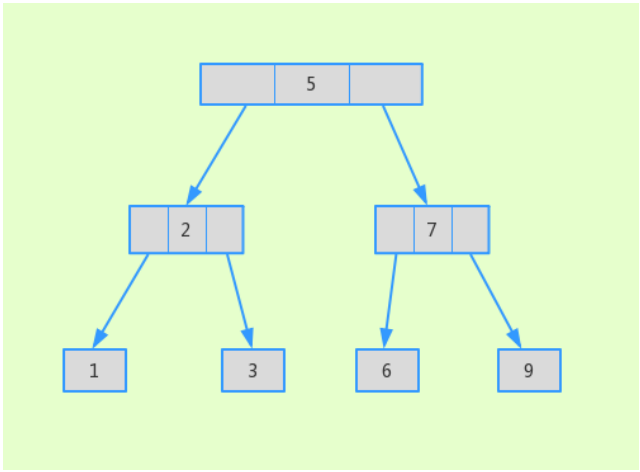
试想一下，如果 3 的右子树后面拖更多的数字，那查询效率得多低啊！

因此，如果想让二叉查找树性能最好，就需要这颗树是平衡的，此时，平衡二叉树出场了。

1.3 平衡二叉树

我们一起看下平衡二叉树的定义：满足二叉查找树的定义，另外必须满足任何节点的两个子树的高度差最大为 1。

比如我们要从 1、2、3、5、6、7、9 几个数字中找到某一个数字，如果使用二叉树进行查找，则结构如下：



如果要查值为 6 的记录，先找到根（这里是 5 ），这里借用二分查找的思想，因为要找的值 6 大于中点元素 5，所以需要查找的是 5 的右子树，而又因为 6 小于 7，则应该找 7 的左子树，找到 6 这条记录，一共查找了 3 次。如果查找记录使用顺序查找，找到 6 这个值需要查 5 次。

对于上面这个例子，我们来计算平衡二叉查找树查询和顺序查询两种方式的平均查找次数：

平衡二叉查找树的平均查找次数：(3 * 4 + 2 * 2 + 1) /7 ≈ 2.4 次

> **这里解释一下为什么这么计算：**
>
> 该平衡二叉查找树中 4 个第三层的值需要查找 3 次，2 个第二层的值需要查找 2 次，第一层也就是根的值只需要查 1 次。

顺序查找的平均查找次数：(1 + 2 + 3 + 4 + 5 + 6 + 7) /7 = 4 次

显然平衡二叉查找树的平均查找速度比顺序查找更快。

但是平衡二叉树有个缺点就是，每个节点最多只有两个分支，如果数据量比较大，要经历多层节点才能查询在叶子节点的数据。

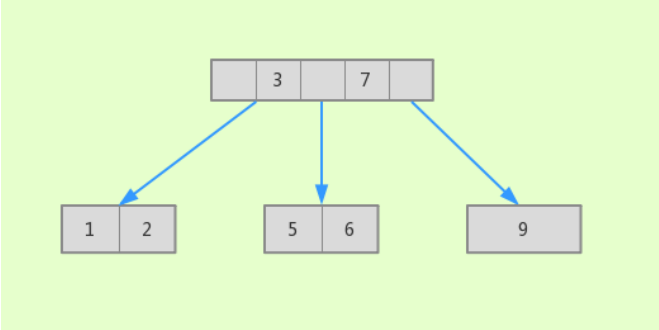
如果在平衡二叉树的基础上，每个节点可以有多个分支，那即使在叶子节点的数据，是不是查询效率也比较高呢？这就引出了 B 树结构。

1.4 B 树

B 树可以理解为一个节点可以拥有多于 2 个子节点的多叉查找树。

B 树中同一键值不会出现多次，要么在叶子节点，要么在内节点上。

比如用 1、2、3、5、6、7、9 这些数字构建一个 B 树结构，其图形如下：



与平衡二叉树相比，B 树利用多个分支（平衡二叉树只有两个分支）节点，减少获取记录时所经历的节点数。

B 树也是有缺点的，因为每个节点都包含 key 值和 data 值，因此如果 data 比较大时，每一页存储的 key 会比较少；当数据比较多时，同样会有：“要经历多层节点才能查询在叶子节点的数据”的问题。这时，B+ 树站了出来。

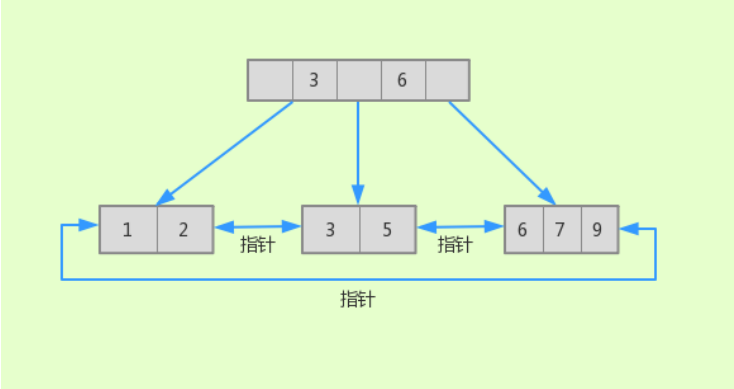
1.5 B+ 树

B+ 树是 B 树的变体，定义基本与 B 树一致，与 B 树的不同点：

- 所有叶子节点中包含了全部关键字的信息
- 各叶子节点用指针进行连接
- 非叶子节点上只存储 key 的信息，这样相对 B 树，可以增加每一页中存储 key 的数量。
- B 树是纵向扩展，最终变成一个“瘦高个”，而 B+ 树是横向扩展的，最终会变成一个“矮胖子”（这里参考了《MySQL 运维内参》第 8 节 B+ 树及 B 树的区别中的比喻）。

在 B+ 树中，所有记录节点都是按键值的大小顺序存放在同一层的叶子节点上。B+ 树中的 B 不是代表二叉(binary)而是代表（balance），B+ 树并不是一个二叉树。

还是根据前面提到的这组数字（1、2、3、5、6、7、9）举例，它的结构如下：



与 1.4 中 B 树的结构最大的区别就是：

它的键一定会出现在叶子节点上，同时也有可能在非叶子节点中重复出现。而 B 树中同一键值不会出现多次。

2 B+ 树索引

在上面的内容中，我们了解到跟索引相关的一些算法。这里就来到了本节的重点内容：B+ 树索引。

B+ 树索引就是基于本节前面介绍的 B+ 树发展而来的。在数据库中，B+ 树的高度一般都在 2 ~ 4 层，所以**查找某一行数据最多只需要 2 到 4 次 IO。而没索引的情况，需要逐行扫描，明显效率低很多，这也就是为什么添加索引能提高查询速度。**

B+ 树索引并不能找到一个给定键值的具体行，B+ 树索引能找到的只是被查找数据行所在的页。然后数据库通过把页读入到缓冲池（buffer pool）中，在内存中通过二分查找法进行查找，得到需要的数据。

InnoDB 中 B+ 树索引分为聚集索引和辅助索引，我们再继续了解这两种索引的特点。

为了方便理解，我们先创建一张测试表并写入数据：

```
use muke; /* 使用muke这个database */
drop table if exists t8; /* 如果表t1存在则删除表t1 */
CREATE TABLE `t8` (
`id` int(11) NOT NULL AUTO_INCREMENT,
`a` int(11) NOT NULL,
`b` char(2) NOT NULL,
PRIMARY KEY (`id`),
KEY `idx_a` (`a`)
) ENGINE=InnoDB  DEFAULT CHARSET=utf8mb4;

insert into t8(a,b) values (1,'a'),(2,'b'),(3,'c'),(5,'e'),(6,'f'),(7,'g'),(9,'i');
```

2.1 聚集索引

InnoDB 的数据是按照主键顺序存放的，而聚集索引就是按照每张表的主键构造一颗 B+ 树，它的叶子节点存放的是整行数据。

InnoDB 的主键一定是聚集索引。如果没有定义主键，聚集索引可能是第一个不允许为 null 的唯一索引，也有可能是 row id。

由于实际的数据页只能按照一颗 B+ 树进行排序，因此每张表只能有一个聚集索引（TokuDB 引擎除外）。查询优化器倾向于采用聚集索引，因为聚集索引能够在 B+ 树索引的叶子节点上直接找到数据。

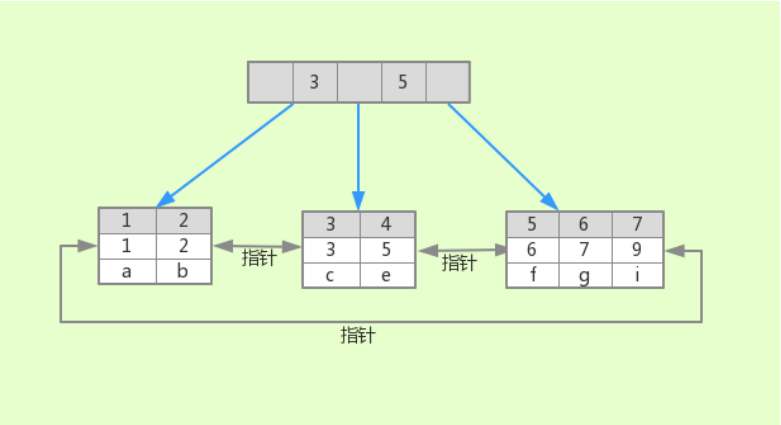
聚集索引对于主键的排序查找和范围查找速度非常快。

对于刚刚创建好的测试表 t8，我们先查询下表的所有数据：

```
select * from t8;
```

```
mysql> select * from t8;
+----+---+---+
| id | a | b |
+----+---+---+
|  1 | 1 | a |
|  2 | 2 | b |
|  3 | 3 | c |
|  4 | 5 | e |
|  5 | 6 | f |
|  6 | 7 | g |
|  7 | 9 | i |
+----+---+---+
7 rows in set (0.00 sec)
```

表 t8 的聚集索引的大致结构如下：



两点关键信息：

- 根据主键值创建了 B+ 树结构
- 每个叶子节点包含了整行数据

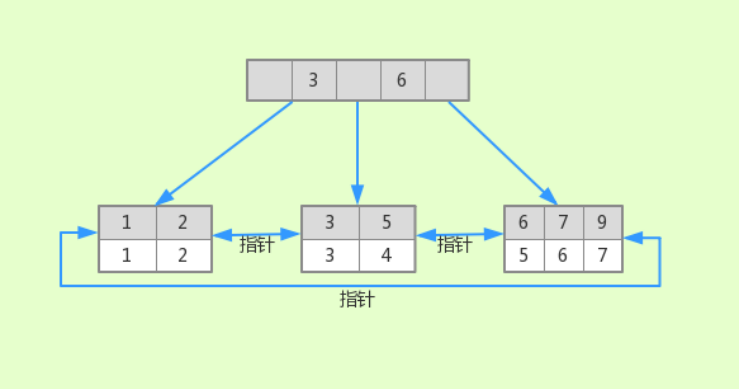
2.2 辅助索引

我们现在知道了聚集索引的叶子节点存放了整行数据，而 InnoDB 存储引擎辅助索引的叶子节点并不会放整行数据，而存放的是键值和主键 ID。

当通过辅助索引来寻找数据时，InnoDB 存储引擎会遍历辅助索引树查找到对应记录的主键，然后通过主键索引来找到对应的行数据。

比如一颗高度为 3 的辅助索引树中查找数据，那需要对这颗辅助索引树遍历 3 次找到指定主键，如果聚集索引树的高度也为 3，那么还需要对聚集索引树进行 3 次查找，最终找到一个完整的行数据所在的页，因此获取数据一共需要6次逻辑 IO 访问。

我们继续拿表 t8 分析，它的辅助索引 idx_a 结构如下：



上图中两点关键点需要注意：

- 根据 a 字段的值创建了 B+ 树结构
- 每个叶子节点保存的是 a 字段自己的键值和主键 ID

对于表 t8，比如有下面这条查询语句：

```
select * from t8 where a=3;
```

它先通过 a 字段上的索引树，得到主键 id 为 3，再到 id 的聚集索引树上找到对应的行数据。

而下面这条 SQL:

```
select * from t8 where id=3;
```

查询到的结果是一样的，而执行过程则只需要搜索 id 的聚集索引树。我们能看出辅助索引的查询比主键查询多扫描一颗索引树，所以，我们应该**尽量使用主键做为条件进行查询**。

3 总结

MySQL 中，使用最多的就是 B+ 树索引，而 B+ 树索引由 B+ 树发展而来。要想理解 B+ 树，需要先了解二分查找法、二叉查找树、平衡二叉树、B 树的一些思想。本节使用了在一组数字中查找某个值的例子，一一说明了各个算法的特点。

在后面的内容中，我们又介绍了 InnoDB 中 B+ 树索引的两种类型：聚集索引和辅助索引，并介绍了它们的实现方式。

相信通过本节学习能让你对索引有进一步的了解。

4 问题

数据量相同的情况下，B 树和 B+ 树，哪个占用空间更大？

5 参考资料

《MySQL 技术内幕》第 2 版 第 5 章 索引与算法

《MySQL 运维内参》 第 8 节 InnoDB 索引实现原理

```
}
```

09 为何count(*)这么慢?

11 哪些情况需要添加索引?