

Crazyfzw's blog



Return to base, step by step.

# 数据存储 1：数据库索引的原理及使用策略

📅 2018-07-18 | 📅 2021-01-02 | 📁 RDBMS | 👁 1164 | 💬 0

📄 13k | ⌚ 12 分钟

本文以 MySQL 数据库为研究对象，讨论与数据库索引相关的一些话题。特别需要说明的是，MySQL支持诸多存储引擎，而各种存储引擎对索引的支持也各不相同，因此MySQL数据库支持多种索引类型，如BTree索引，哈希索引，全文索引等等。为了避免混乱，本文将只关注于BTree索引，因为这是平常使用MySQL时最经常用到的索引。

## 一、数据库索引的原理

这部分内容主要来源于互联网上关于索引的主流文章，本人在理解的基础上梳理整合成本部分内容，当是个人的一篇学习笔记，参考的文章可见参考文献一栏。



## 1. 索引的本质

MySQL官方对索引的定义为：索引（Index）是帮助数据库高效获取数据的数据结构。提取句子主干，就可以得到索引的本质：索引是数据结构。

说白了，索引就是一种可以应用高效查询算法的数据结构。

之所以要建立索引，其实就是为了构建一种数据结构，然后可以在上面应用一种高效的查询算法，最终提高数据的查询速度。

所以在讲索引的数据结构之前不妨先了解下比较常见的查询算法及其所要求的特定数据结构。

## 2. 常见的查询算法

### 2.1 顺序查找（linear search）

最基本的查询算法当然是顺序查找（linear search），也就是对比每个元素的方法，不过这种算法在数据量大时效率极低。

数据结构：有序或无序队列

复杂度： $O(n)$

实例代码：



```
1 //顺序查找
2 int SequenceSearch(int a[], int value, int n)
3 {
4     int i;
5     for(i=0; i<n; i++)
6         if(a[i]==value)
7             return i;
8     return -1;
9 }
```


## 2.2 二分查找 (binary search)

比顺序查找更快的查询方法应该就是二分查找了，二分查找的原理是查找过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜索过程结束；如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。如果在某一步骤数组为空，则代表找不到。

数据结构：有序数组

复杂度： $O(\log n)$

实例代码：



```
1 //二分查找，递归版本
2 int BinarySearch2(int a[], int value, int low, int high)
3 {
4     int mid = low+(high-low)/2;
```

```
5     if(a[mid]==value)
6         return mid;
7     if(a[mid]>value)
8         return BinarySearch2(a, value, low, mid-1);
9     if(a[mid]<value)
10        return BinarySearch2(a, value, mid+1, high);
11 }
```

## 2.3 二叉排序树查找

二叉排序树的特点是：

若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；

若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；

它的左、右子树也分别为二叉排序树。

搜索的原理：

若b是空树，则搜索失败，否则；

若x等于b的根节点的数据域之值，则查找成功；否则：

若x小于b的根节点的数据域之值，则搜索左子树；否则：

查找右子树。

数据结构：二叉排序树



时间复杂度:  $O(\log_2 N)$

## 2.4 索引是为了应用某种高级查找算法而建立的一种数据结构:

稍微分析以上查找算法会发现, 每种查找算法都只能应用于特定的数据结构之上, 例如二分查找要求被检索数据有序, 而二叉树查找只能应用于二叉查找树上。但是数据库存储的数据本身的组织结构不可能完全满足各种数据结构 (例如, 理论上不可能同时将两列都按顺序进行组织), 所以, 在数据之外, 数据库系统还维护着满足特定查找算法的数据结构, 这些数据结构以某种方式引用 (指向) 数据, 这样就可以在这些数据结构上实现高级查找算法。这种数据结构, 就是索引。

## 3. 多叉平衡查找树 (B-tree 和 B+Tree)

上面讲到了二叉树, 它的搜索时间复杂度为 $O(\log_2 N)$ , 所以它的搜索效率和树的深度有关, 如果要提高查询速度, 那么就要降低树的深度。要降低树的深度, 很自然的方法就是采用多叉树, 再结合平衡二叉树的思想, 我们可以构建一个平衡多叉树结构, 然后就可以在上面构建平衡多路查找算法, 提高大数据量下的搜索效率。

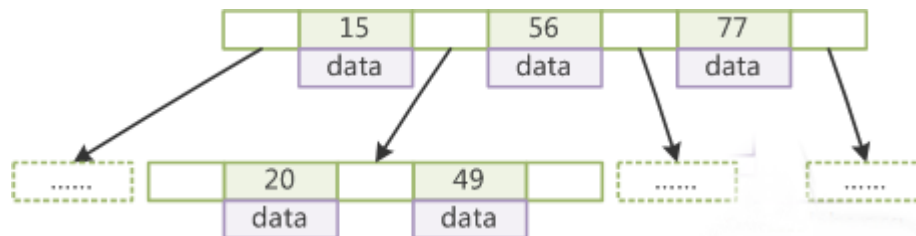
### 3.1 B-Tree

具体讲解之前, 有一点, 再次强调下: B-树, 也就是B树。因为B树的原英文名称为B-tree, 而国内很多人喜欢把B-tree译作B-树, 其实, 这是个非常不好的直译, 很容易让人产生误解, 有人可能会以为B-树是一种树, 而B树又是另一种树。而事实上是, B-tree指的就是B树。特此说明。



为了描述B-Tree, 首先定义一条数据记录为一个二元组[key, data], key为记录的键值, 对于不同数据记录, key是互不相同的; data为数据记录除key外的数据。

下图是一个B-Tree示例:



那么B-Tree是满足下列条件的数据结构:

d为大于1的一个正整数, 称为B-Tree的度。

h为一个正整数, 称为B-Tree的高度。

每个非叶子节点由n-1个key和n个指针组成, 其中 $d \leq n \leq 2d$ 。

每个叶子节点最少包含一个key和两个指针, 最多包含 $2d-1$ 个key和 $2d$ 个指针, 叶节点的指针均为null。

所有叶节点具有相同的深度, 等于树高h。

key和指针互相间隔, 节点两端是指针。

一个节点中的key从左到右非递减排列。

所有节点组成树结构。

每个指针要么为null，要么指向另外一个节点。

如果某个指针在节点node最左边且不为null，则其指向节点的所有key小于v(key1)，其中v(key1)为node的第一个key的值。

如果某个指针在节点node最右边且不为null，则其指向节点的所有key大于v(keym)，其中v(keym)为node的最后一个key的值。

如果某个指针在节点node的左右相邻key分别是keyi和keyi+1且不为null，则其指向节点的所有key小于v(keyi+1)且大于v(keyi)。

由于B-Tree的特性，在B-Tree中按key检索数据的算法非常直观：首先从根节点进行二分查找，如果找到则返回对应节点的data，否则对相应区间的指针指向的节点递归进行查找，直到找到节点或找到null指针，前者查找成功，后者查找失败。B-Tree上查找算法的伪代码如下：

```
1 BTree_Search(node, key) {
2     if(node == null) return null;
3     foreach(node.key){
4         if(node.key[i] == key) return node.data[i];
5         if(node.key[i] > key) return BTree_Search(point[i]->node);
6     }
7     return BTree_Search(point[i+1]->node);
```



```
8     }  
9     data = BTree_Search(root, my_key);
```

关于B-Tree有一系列有趣的性质，例如一个度为 $d$ 的B-Tree，设其索引 $N$ 个key，则其树高 $h$ 的上限为 $\log_d((N+1)/2)$ ，检索一个key，其查找节点个数的渐进复杂度为 $O(\log_d N)$ 。从这点可以看出，B-Tree是一个非常有效率的索引数据结构。

另外，由于插入删除新的数据记录会破坏B-Tree的性质，因此在插入删除时，需要对树进行一个分裂、合并、转移等操作以保持B-Tree性质，这也是索引会降低增删改数据性能的原因。

### 3.2 B+Tree

B-Tree有许多变种，其中最常见的是B+Tree，例如MySQL就普遍使用B+Tree实现其索引结构。

与B-Tree相比，B+Tree有以下不同点：

每个节点的指针上限为 $2d$ 而不是 $2d+1$ 。

所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接。（而B树的叶子节点并没有包括全部需要查找的信息）

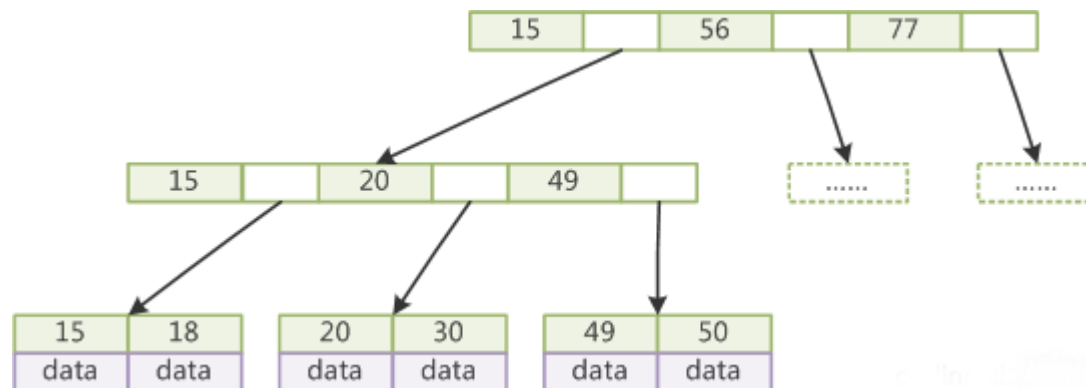
内节点不存储data，只存储key；叶子节点不存储指针。所有的内节点点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。（而B树的内节点也包含需要查





找的有效信息)

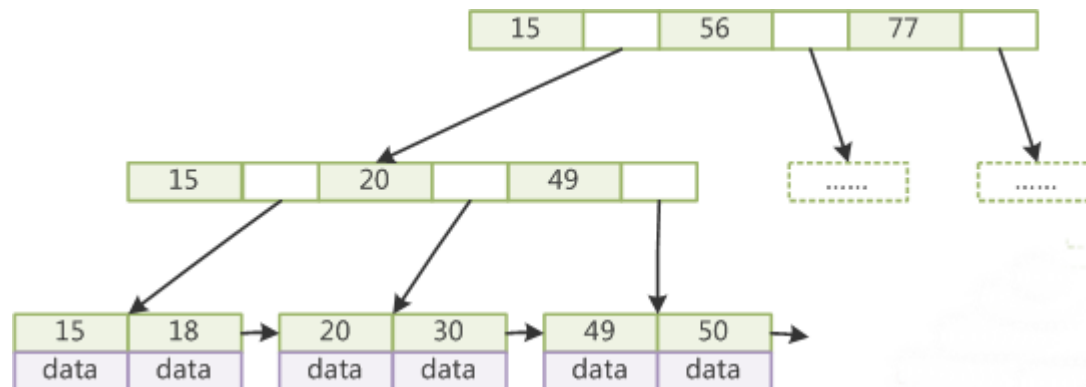
下图是一个简单的B+Tree示意



## 4. 为什么索引用B+Tree实现

这里必须强调的是，数据库索引实际上用的是带有顺序的B+Tree。在经典B+Tree的基础上进行了优化，增加了顺序访问指针。

如下图所示：



如图所示，在B+Tree的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问指针的B+Tree。做这个优化的目的是为了提高区间访问的性能。如图如果要查询key为从18到49的所有数据记录，当找到18后，只需顺着节点和指针顺序遍历就可以一次性访问到所有数据节点，极大提高了区间查询效率。

#### 4.1 为什么索引采用B+Tree/B-Tree的性能会比采用红黑树好？

上文说过一般使用磁盘I/O次数评价索引结构的优劣。先从B-Tree分析，根据B-Tree的定义，可知检索一次最多需要访问h个节点。数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现B-Tree还需要使用如下技巧：

每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

B-Tree 中一次检索最多需要  $h-1$  次 I/O（根节点常驻内存），渐进复杂度为  $O(h)=O(\log dN)$ 。一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。

综上所述，用B-Tree作为索引结构效率是非常高的。

而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的I/O渐进复杂度也为 $O(h)$ ，效率明显比B-Tree差很多。

#### 4.2 为什么索引采用B+Tree的性能会比采用B-Tree好？

[↑](#)

B+Tree更适合外存索引，原因和内节点出度d有关。从上面分析可以看到，d越大索引的性能越好，而出度的上限取决于节点内key和data的大小：

```
1 dmax=floor(pagesize/(keysize+datasize+pointsize))
```

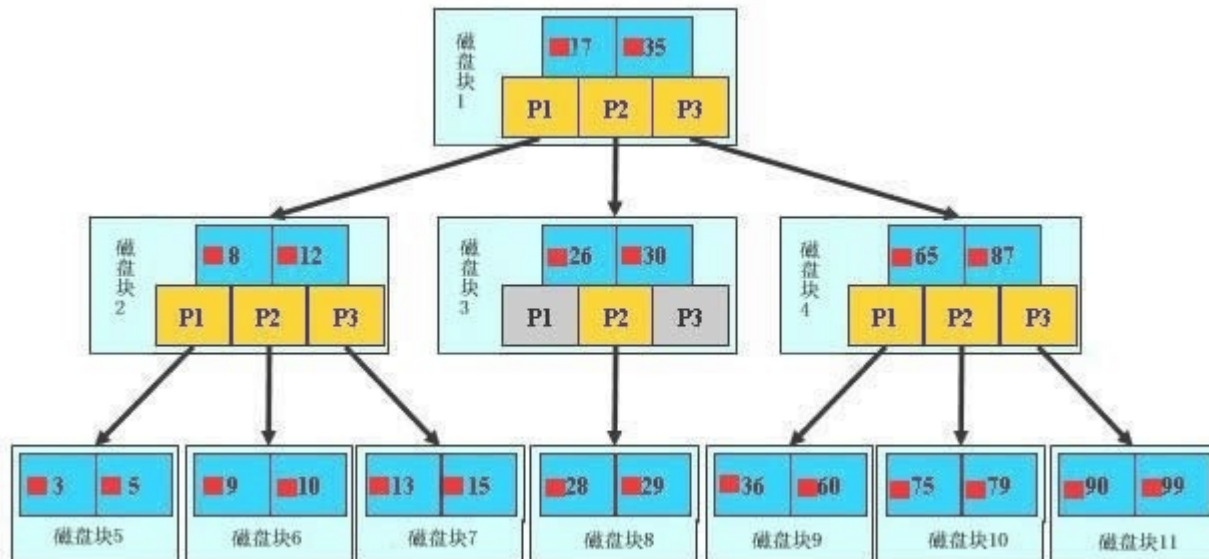
floor表示向下取整。由于B+Tree内节点去掉了data域，因此可以拥有更大的出度，拥有更好的性能。

因为B+Tree 的内节点只存储只存储key，不存储Data，所以出度越大，索引性能越好。

## 5. B+Tree 的查找过程

B-树和B+树查找过程基本一致。

下面以 B+Tree 树的数据结构举例说下查找过程：



如上图 B + Tree 的数据结构。是由一个一个的磁盘块组成的树形结构，每个磁盘块由数据项和指针组成。所有的数据都是存放在叶子节点，非叶子节点不存放数据。

### 查找过程：

以磁盘块1为例，指针 P1 表示小于17的磁盘块，P2 表示在 17~35 之间的磁盘块，P3 则表示大于35的磁盘块。

比如要查找数据项99，首先将磁盘块1 load 到内存中，发生 1 次 IO。接着通过二分查找发现 99 大于 35，所以找到了 P3 指针。通过P3 指针发生第二次 IO 将磁盘块4加载到内存。再通过二分查找发现大于87，通过 P3 指针发生了第三次 IO 将磁盘块11 加载到内存。最后再通过一次二分查找找到了数据项99。

由此可见，如果一个几百万的数据查询只需要进行三次 IO 即可找到数据，那么整个效率将是非常高的。



观察树的结构，发现查询需要经历几次 IO 是由树的高度来决定的，而树的高度又由磁盘块，数据项的大小决定的。

磁盘块越大，数据项越小那么数的高度就越低。这也就是为什么索引字段要尽可能小的原因。

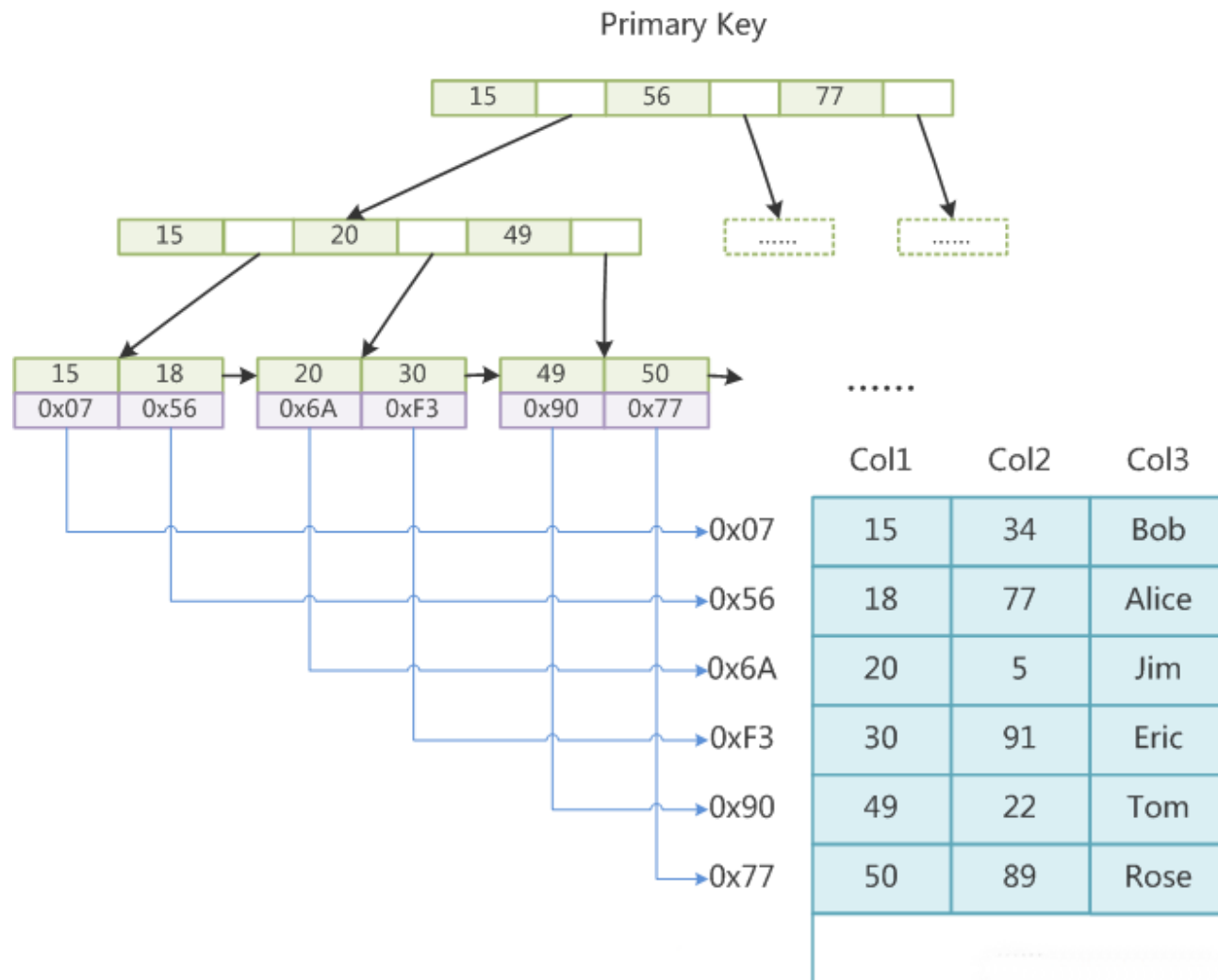
## 6. MySQL索引实现

在MySQL中，索引属于存储引擎级别的概念，不同存储引擎对索引的实现方式是不同的，本文主要讨论MyISAM和InnoDB两个存储引擎的索引实现方式。

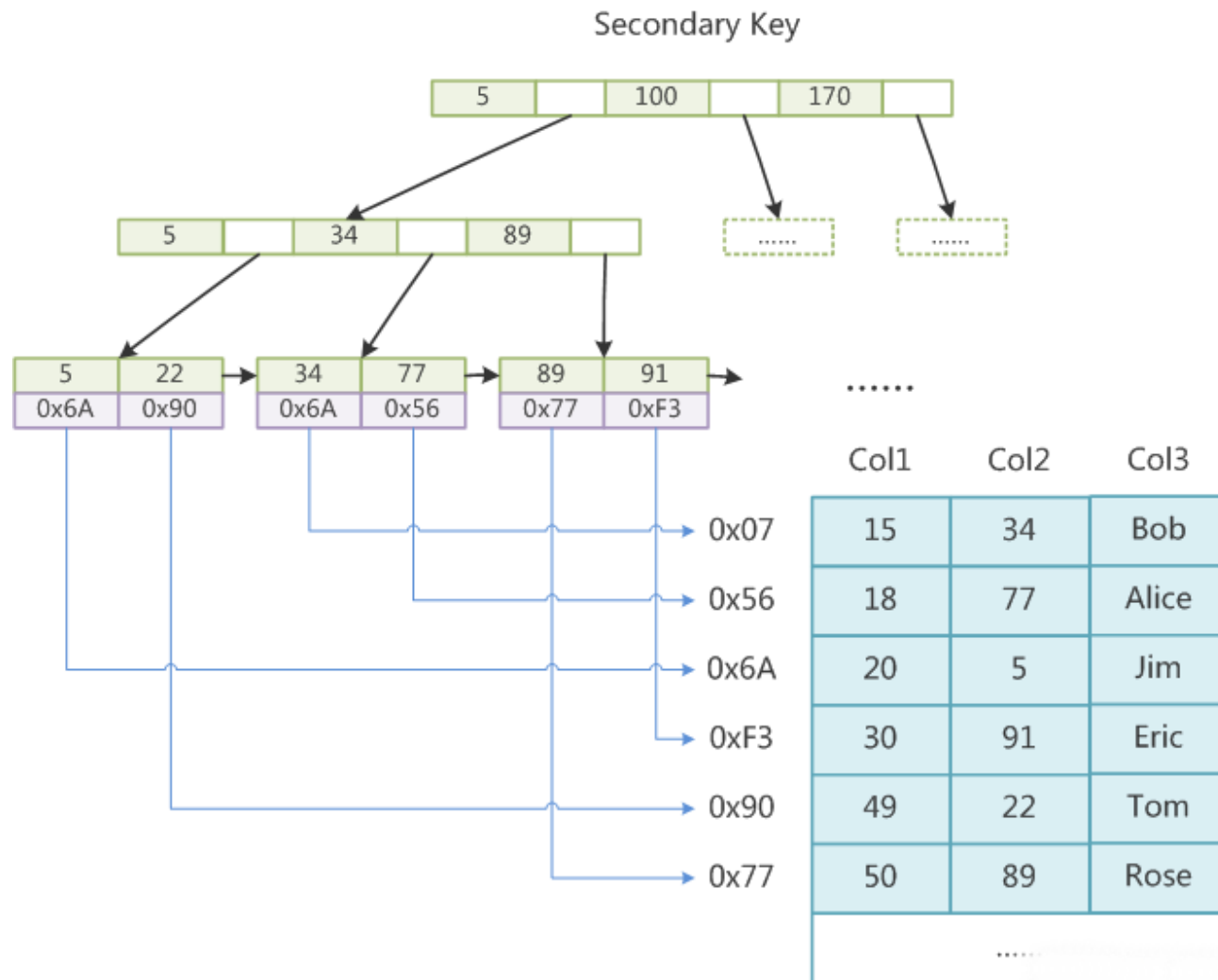
### 6.1 MyISAM索引实现

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。数据存储在—个地方，索引存储在另一个地方，索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同，这种索引方式称为“非聚集索引”。

下图是MyISAM索引的原理图：



这里设表一共有三列，假设我们以Col1为主键，上图是一个MyISAM表的主索引（Primary key）示意。可以看出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，只是主索引要求key是唯一的，而辅助索引的key可以重复。如果我们在Col2上建立一个辅助索引，则此索引的结构如下图所示：



同样也是一颗B+Tree，data域保存数据记录的地址。因此，MyISAM中索引检索的算法为首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其data域的值，然后以data域的值作为地址，读取相应数据记录。

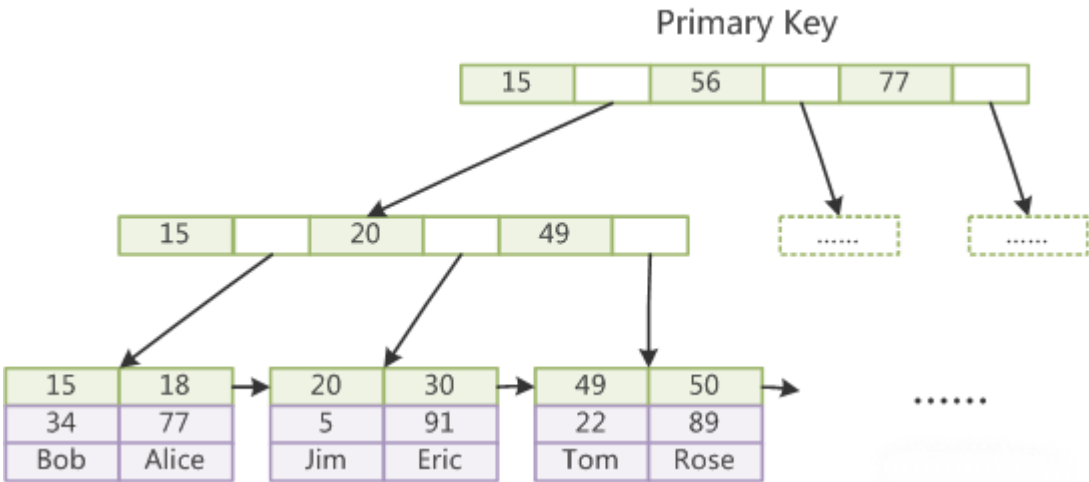
MyISAM的索引方式也叫做“非聚集”的，之所以这么称呼是为了与InnoDB的聚集索引区分。

## 6.2 InnoDB索引实现

虽然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。

第一个重大区别是InnoDB的数据文件本身就是索引文件。从上文知道，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组织的一个索引结构，这棵树的叶节点data域保存了完整的数据记录。这个索引的key是数据表的主键，因此InnoDB表数据文件本身就是主索引。键值的逻辑顺序决定了表中相应行的物理顺序，这种索引方式称为“聚集索引”。

如下图所示：

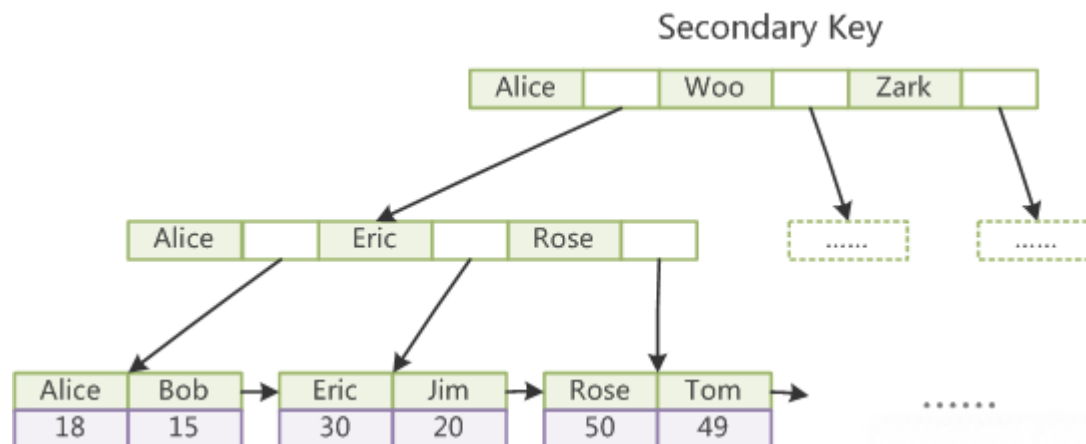


上图是InnoDB主索引（同时也是数据文件）的示意图，可以看到叶节点包含了完整的数据记录。这种索引叫做聚集索引。因为InnoDB的数据文件本身要按主键聚集，所以InnoDB要求表必须有主键（MyISAM可以没有），如果没有显式指定，则MySQL系统会自动选择一个可以唯一标识数据记录的列作为主键，如果不存在这种列，则MySQL自动为InnoDB表生成一个隐含字段作为主键，这个字段长度为6个字节，类型为长整形。





第二个与MyISAM索引的不同是InnoDB的辅助索引data域存储相应记录主键的值而不是地址。换句话说，InnoDB的所有辅助索引都引用主键作为data域。例如，下图为定义在Col3上的一个辅助索引：



这里以英文字符的ASCII码作为比较准则。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，然后用主键到主索引中检索获得记录。

了解不同存储引擎的索引实现方式对于正确使用和优化索引都非常有帮助，例如知道了InnoDB的索引实现后，就很容易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索引变得过大。再例如，用非单调的字段作为主键在InnoDB中不是个好主意，因为InnoDB数据文件本身是一颗B+Tree，非单调的主键会造成在插入新记录时数据文件为了维持B+Tree的特性而频繁的分裂调整，十分低效，而使用自增字段作为主键则是一个很好的选择。

## 7. 聚集索引与非聚集索引的区别



前面已经通过MySQL索引实现中已经讲到InnoDB索引实现用的是“聚集索引”，而MyISAM索引实现的实现用的是“非聚集索引”，那么 聚集索引与非聚集索引到底有什么区别呢？

**聚集索引和非聚集索引的根本区别是表记录的排列顺序和与索引的排列顺序是否一致。**

## 7.1 聚集索引

聚集索引表记录的排列顺序与索引的排列顺序一致

**优点：**查询速度快，因为一旦具有第一个索引值的纪录被找到，具有连续索引值的记录也一定物理的紧跟其后。

**缺点：**对表进行修改速度较慢，这是为了保持表中的记录的物理顺序与索引的顺序一致，而把记录插入到数据页的相应位置，必须在数据页中进行数据重排，从而降低了执行速度。

## 7.2 非聚集索引

叶节点的data域存放的是数据记录的地址。数据存储在一个地方，索引存储在另一个地方，索引中索引的逻辑顺序与磁盘上行的物理存储顺序不同。

**优点：**对数据更新影响较小。

**缺点：**聚集索引检索效率比非聚集索引低，必须先查到查到每一项数据对应的页码，然后再根据页码查到具体内容。

7.3 如何选择使用聚集索引或非聚集索引（很重要）：

动作描述	使用聚集索引	使用非聚集索引
列经常被分组排序	应	应
返回某范围内的数据	应	不应
一个或极少不同值	不应	不应
小数目的不同值	应	不应
大数目的不同值	不应	应
频繁更新的列	不应	应
外键列	应	应
主键列	应	应
频繁修改索引列	不应	应

二、如何设计索引

既然索引可以加快查询速度，那么是不是只要是查询语句需要，就建上索引？答案是否定的。因为索引虽然加快了查询速度，但索引也是有代价的：索引文件本身要消耗存储空间，同时索引会加重插入、删除和修改记录时的负担，另外，数据库在运行时也要消耗资源维护索引，因此索引并不是越多越好。



## 1. 一般两种情况下不建议建索引:

第一种情况是表记录比较少, 例如一两千条甚至只有几百条记录的表, 没必要建索引, 让查询做全表扫描就好了。至于多少条记录才算多, 这个个人有个人的看法, 我个人的经验是以2000作为分界线, 记录数不超过 2000可以考虑不建索引, 超过2000条可以酌情考虑索引。

另一种不建议建索引的情况是索引的选择性较低。所谓索引的选择性 (Selectivity), 是指不重复的索引值 (也叫基数, Cardinality) 与表记录数 (#T) 的比值:

1     $\text{Index Selectivity} = \text{字段对应不重复的值} / \text{表总记录数}$

显然选择性的取值范围为(0, 1], 值越大, 即选择性越高的索引价值越大, 这是由B+Tree的性质决定的。

例如, 可以应用求一个字段的Selectivity, 判断是否有必要给这个这个字段加索引, 以及

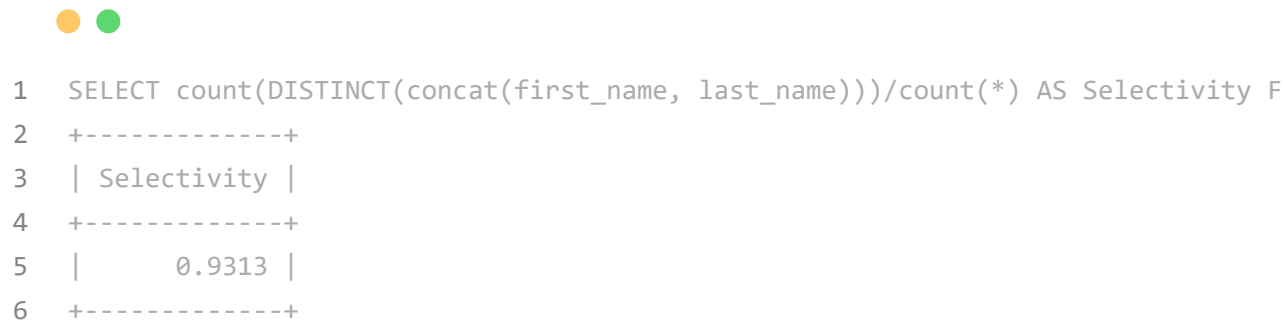
```
1  SELECT count(DISTINCT(title))/count(*) AS Selectivity FROM Table1;
2  +-----+
3  | Selectivity |
4  +-----+
5  |      0.0000 |
6  +-----+
```

title的选择性不足0.0001（精确值为0.00001579），所以实在没有什么必要为其单独建索引。

## 2. 设计索引一般需要考虑以下3点：

1. 看数据量，根据记录数，看是否有建索引的必要；
2. 根据计算字段的索引选择性判断给某个字段加索引是否比较有价值；
3. 看能否使用前缀索引取代全列索引，综合考虑索引选择性与key的长度，做个折中，尽可能使得前缀索引的选择性接近全列索引，同时又减短索引key的长度，从而减少了索引文件的大小和维护开销。

举个例子：



```
1  SELECT count(DISTINCT(concat(first_name, last_name)))/count(*) AS Selectivity F
2  +-----+
3  | Selectivity |
4  +-----+
5  |      0.9313 |
6  +-----+
```

<first\_name, last\_name> 选择性很好，但是first\_name和last\_name加起来长度为30，有没有兼顾长度和选择性的办法？可以考虑用first\_name和last\_name的前几个字符建立索引，例如<first\_name, left(last\_name, 3)>，看看其选择性：

```

1  SELECT count(DISTINCT(concat(first_name, left(last_name, 3))))/count(*) AS Sele
2  +-----+
3  | Selectivity |
4  +-----+
5  |      0.7879 |
6  +-----+

```

择性还不错，但离0.9313还是有点距离，那么把last\_name前缀加到4:

```

1  SELECT count(DISTINCT(concat(first_name, left(last_name, 4))))/count(*) AS Sele
2  +-----+
3  | Selectivity |
4  +-----+
5  |      0.9007 |
6  +-----+

```

这时选择性已经很理想了，而这个索引的长度只有18，比<first\_name, last\_name>短了接近一半，我们把这个前缀索引建上:

```

1  ALTER TABLE Table1
2  ADD INDEX `first_name_last_name4` (first_name, last_name(4));

```

此时再执行一遍按名字查询，比较分析一下与建索引前的结果:



```
1  SHOW PROFILES;
2  +-----+-----+-----+
3  | Query_ID | Duration | Query
4  +-----+-----+-----+
5  |          87 | 0.11941700 | SELECT * FROM Table1 WHERE first_name='Eric' AND last
6  |          90 | 0.00092400 | SELECT * FROM Table1 WHERE first_name='Eric' AND last
```

性能的提升是显著的，查询速度提高了120多倍。

有一点需要注意的是：前缀索引兼顾索引大小和查询速度，但是其缺点是不能用于ORDER BY和GROUP BY操作，也不能用于Covering index（即当索引本身包含查询所需全部数据时，不再访问数据文件本身）。

### 三、索引使用的注意事项(策略及优化)

并不是建立索引就能显著提高查询速度，在索引的使用过程中，存在一些使用细节和注意事项，因为稍不留心，就可能导致在查询过程中索引失效。

一下列举一些需要注意的事项：

#### 1. 不要在列上使用函数

不要在列上使用函数，这将导致索引失效而进行全表扫描。

如：



```
1 select * from news where year(publish_time) < 2018
```

应改为:

```
1 select * from news where publish_time < '2018-01-01'
```

## 2. 不要在列上进行计算

不要在列上进行运算，这也将导致索引失效而进行全表扫描。

如:

```
1 select * from news where id / 100 = 1
```

应改为:

```
1 select * from news where id = 1 * 100
```

## 3. 尽量避免使用 != 或 not in 或 <> 等否定操作符

应该尽量避免在 where 子句中使用 != 或 not in 或 <> 操作符，这些负向查询也会导致索引失效而进行全表扫描。



如:



```
1 select name from user where id not in (1,3,4);
```

应改为:



```
1 select name from user where id in (2,5,6);
```

## 4. 尽量避免使用 or 来连接条件

应该尽量避免在 where 子句中使用 or 来连接条件, 因为这会导致索引失效而进行全表扫描。

如:



```
1 select * from CRM_CUSTOMER_INFO where id= 1 or id =2
```

应改为



```
1 select * from CRM_CUSTOMER_INFO where id in(1,2)
```

## 5. 字段的默认值不要为 null

只要列中包含有 NULL 值都将不会被包含在索引中，复合索引中只要有一列含有 NULL 值，那么这一列对于此复合索引就是无效的。

因此，在数据库设计时，除非有一个很特别的原因使用 NULL 值，不然尽量不要让字段的默认值为 NULL。

## 6. 不要让数据库帮我们做隐式类型转换

当查询条件左右两侧类型不匹配的时候会发生隐式转换，隐式转换带来的影响就是可能导致索引失效而进行全表扫描。



```
1 select name from user where telno=1888888888
```

这样虽然可以查出数据，但是会导致全表扫描。

应改为



```
1 select name from user where telno='1888888888'
```

## 7. 前导模糊查询会导致索引失效

like 的方式进行查询，在 like “value%” 可以使用索引，但是对于 like “%value%” 这样的方式，执行全表查询，这在数据量小的表，不存在性能问题，但是对于海量数据，全表扫描是非常可怕的事情。所以，根据业务需求，考虑使用 Elasticsearch 或 Solr 是个不错的方案。

## 8. 数据区分不明显的不建议创建索引

如 user 表中的性别字段，可以明显区分的才建议创建索引，如身份证等字段

## 9. 可以用复合索引替代多个单列索引

MySQL 只能使用一个索引，会从多个索引中选择一个限制最为严格的索引，因此，为多个列创建单列索引，并不能提高 MySQL 的查询性能。

假设，有两个单列索引，分别为 news\_year\_idx(news\_year) 和 news\_month\_idx(news\_month)。现在，有一个场景需要针对资讯的年份和月份进行查询，那么，SQL 语句可以写成：

```
select * from news where news_year = 2017 and news_month = 1
```

事实上，MySQL 只能使用一个单列索引。为了提高性能，可以使用复合索引 news\_year\_month\_idx(news\_year, news\_month) 保证 news\_year 和 news\_month 两个列都被索引覆盖。

## 10. 覆盖索引的好处



如果一个索引包含所有需要的查询的字段值，直接根据索引的查询结果返回数据，而无需读表，能够极大的提高性能。因此，可以定义一个让索引包含的额外的列，即使这个列对于索引而言是无用的。

## 11. 范围查询对多列查询的影响

查询中的某个列有范围查询，则其右边所有列都无法使用索引优化查找。

举个例子，假设有一个场景需要查询本周发布的资讯文章，其中的条件是必须是启用状态，且发布时间在这周内。那么，SQL 语句可以写成：

```
select * from news where publish_time >= '2017-01-02' and publish_time <= '2017-01-08' and enable = 1
```

这种情况下，因为范围查询对多列查询的影响，将导致 news\_publish\_idx(publish\_time, enable) 索引中 publish\_time 右边所有列都无法使用索引优化查找。换句话说，news\_publish\_idx(publish\_time, enable) 索引等价于 news\_publish\_idx(publish\_time)。

对于这种情况，我的建议：对于范围查询，务必要注意它带来的副作用，并且尽量少用范围查询，可以通过曲线救国的方式满足业务场景。

例如，上面案例的需求是查询本周发布的资讯文章，因此可以创建一个 news\_weekth 字段用来存储资讯文章的周信息，使得范围查询变成普通的查询，SQL 可以改写成：



```
1 select * from news where news_weekth = 1 and enable = 1
```

然而，并不是所有的范围查询都可以进行改造，对于必须使用范围查询但无法改造的情况，我的建议：不必试图用 SQL 来解决所有问题，可以使用其他数据存储技术控制时间轴，例如 Redis 的 SortedSet 有序集合保存时间，或者通过缓存方式缓存查询结果从而提高性能。

## 12. 复合索引（联合索引）

首先介绍一下联合索引。联合索引其实很简单，相对于一般索引只有一个字段，联合索引可以为多个字段创建一个索引。它的原理也很简单，比如，我们在 (a,b,c) 字段上创建一个联合索引，则索引记录会首先按照A字段排序，然后再按照B字段排序然后再是C字段，因此，联合索引的特点就是：

第一个字段一定是有序的

当第一个字段值相等的时候，第二个字段又是有序的，比如下表中当A=2时所有B的值是有序排列的，依次类推，当同一个B值得所有C字段是有序排列的

A	B	C
1	2	3
1	4	2
1	1	4



A	B	C
2	3	5
2	4	4
2	4	6
2	5	5

其实联合索引的查找就跟查字典是一样的，先根据第一个字母查，然后再根据第二个字母查，或者只根据第一个字母查，但是不能跳过第一个字母从第二个字母开始查。这就是所谓的最左前缀原理。

### 13. 复合索引的最左前缀原理

在复合索引的基础上，再来详细介绍一下联合索引的查询。还是复合索引中的例子，我们在 (a,b,c) 字段上建了一个联合索引，所以这个索引是先按a 再按b 再按c进行排列的，所以：

以下的查询方式都可以用到索引

```
1 select * from table where a=1;
2 select * from table where a=1 and b=2;
3 select * from table where a=1 and b=2 and c=3;
```

上面三个查询按照 (a), (a, b), (a, b, c) 的顺序都可以利用到索引, 这就是最左前缀匹配。

如果查询语句是:



```
1 select * from table where a=1 and c=3; 那么只会用到索引a。
```

如果查询语句是:




```
1 select * from table where b=2 and c=3; 因为没有用到最左前缀a, 所以这个查询是用户到
```



如果用到了最左前缀, 但是顺序颠倒会用到索引吗?

比如:



```
1 select * from table where b=2 and a=1;  
2 select * from table where b=2 and a=1 and c=3;
```

如果用到了最左前缀而只是颠倒了顺序, 也是可以用到索引的, 因为mysql查询优化器会判断纠正这条sql语句该以什么样的顺序执行效率最高, 最后才生成真正的执行计划。但我们还是最好按照索引顺序来查询, 这样查询优化器就不用重新编译了。



## 四、参考文献

[1] MySQL索引背后的数据结构及算法原理 <http://blog.codinglabs.org/articles/theory-of-mysql-index.html>

[2] 从 B 树、B+ 树、B\* 树谈到 R 树 [https://blog.csdn.net/v\\_july\\_v/article/details/6530142](https://blog.csdn.net/v_july_v/article/details/6530142)

[3] 聚集索引和非聚集索引 <https://www.cnblogs.com/aspnethot/articles/1504082.html>

[4] 你知道数据库索引的工作原理吗？ <http://www.ituring.com.cn/article/986>

[5] 如何设计索引 [http://blog.720ui.com/2017/mysql\\_core\\_03\\_how\\_use\\_index/](http://blog.720ui.com/2017/mysql_core_03_how_use_index/)

[# 索引](#) | [# RDBMS](#)

[< 数据存储 2：分布式系统全局唯一ID生成方案](#)

[Java 虚拟机 7：JVM监控与调优 >](#)

昵称	邮箱	网址(http://)
<div>ゞ/≥v≤)o 来呀！吐槽一番吧！</div> <div></div>		







提交

来发评论吧~

Powered By [Valine](#)  
v1.4.14

© 2021 ❤️ crazyfzw | 🏠 545k | ☕ 8:16

由 [Hexo](#) & [NexT.Muse](#) 强力驱动

👤 25836 | 👁 38214

