数据结构与算法（7）：数据库索引原理及优化

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时主要打交道的索引，至于哈希索引和全文索引本文暂不讨论。

文章主要内容分为三个部分。

第一部分主要从数据结构及算法理论层面讨论MySQL数据库索引的数理基础。

第二部分结合MySQL数据库中MyISAM和InnoDB数据存储引擎中索引的架构实现讨论聚集索引、非聚集索引及覆盖索引等话题。

第三部分根据上面的理论基础，讨论MySQL中高性能使用索引的策略。

一、数据结构及算法基础

为什么这里要讲查询算法和数据结构呢？因为之所以要建立索引，其实就是为了构建一种数据结构，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。

1.1 索引的本质

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。提取句子主干，就可以得到索引的本质：索引是数据结构。

我们知道，数据库查询是数据库的最主要功能之一。我们都希望查询数据的速度能尽可能的快，因此数据库系统的设计者会从查询算法的角度进行优化。最基本的查询算法当然是顺序查找（linear search），这种复杂度为O(n)的算法在数据量很大时显然是糟糕的，好在计算机科学的发展提供了很多更优秀的查找算法，例如二分查找（binary search）、二叉树查找（binary tree search）等。

如果稍微分析一下会发现，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。

看一个例子：

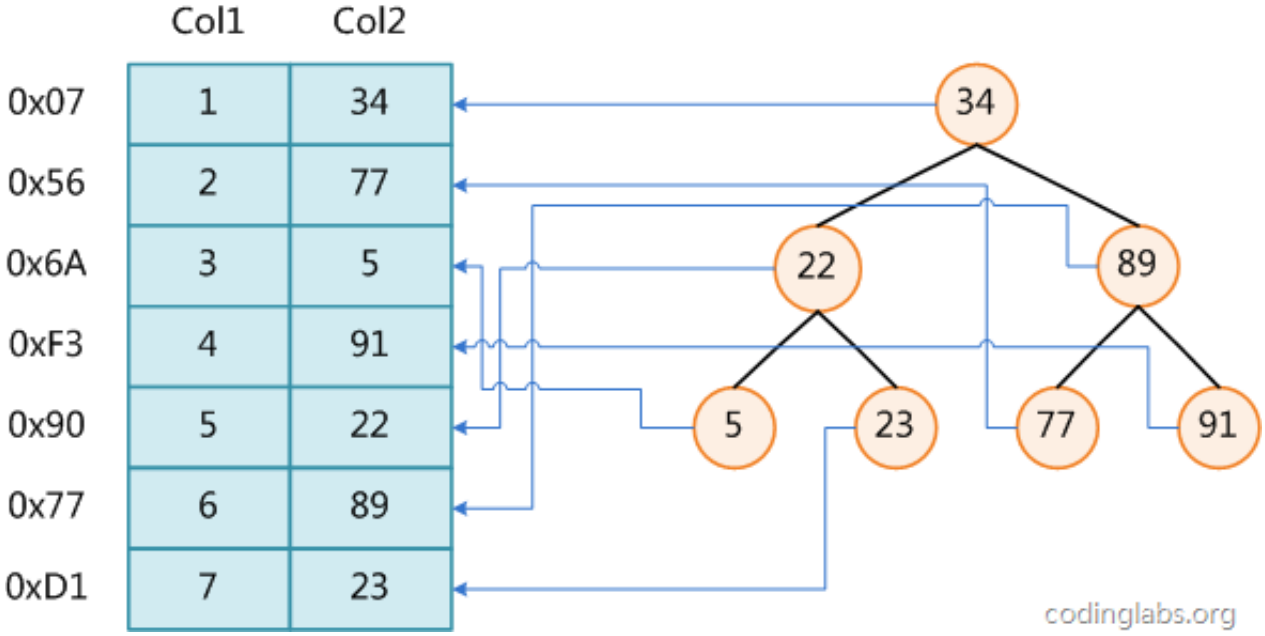


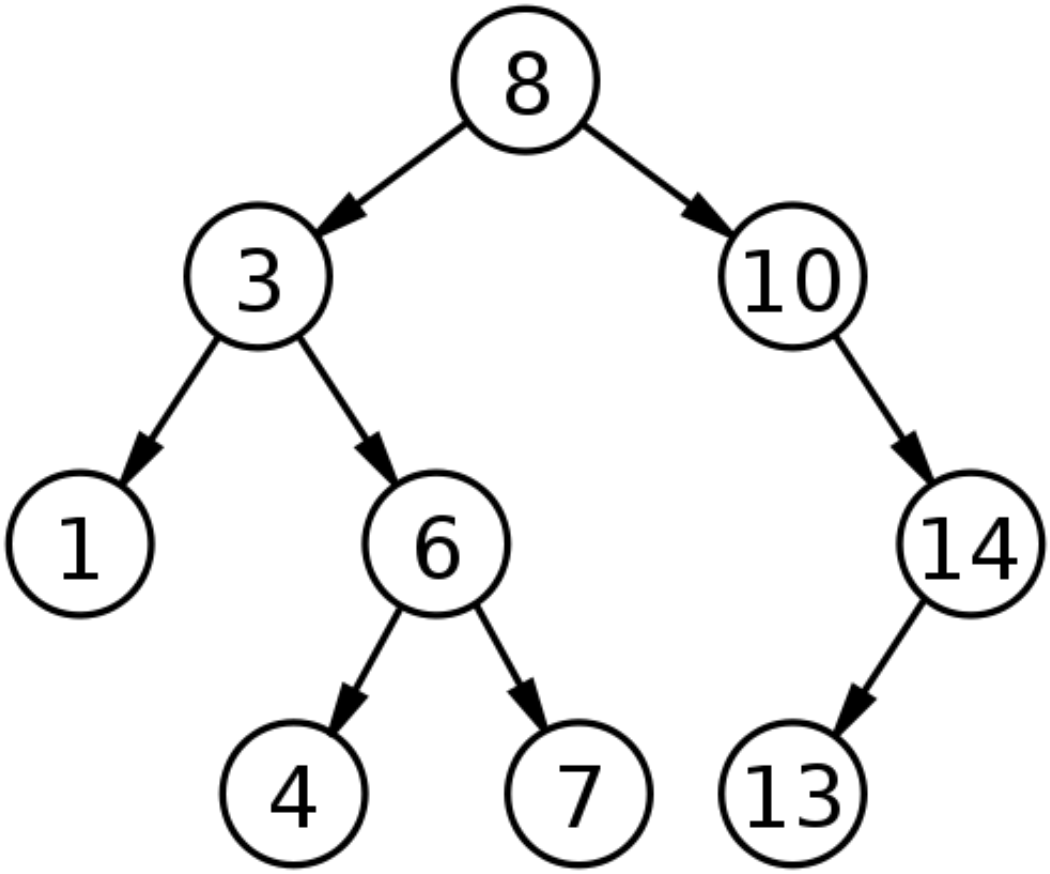
图1展示了一种可能的索引方式。左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址（注意逻辑上相邻的记录在磁盘上也并不是一定物理相邻的）。为了加快Col2的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针，这样就可以运用二叉查找在O(log2n)O(log2n)的复杂度内获取到相应数据。

虽然这是一个货真价实的索引，但是实际的数据库系统几乎没有使用二叉查找树或其进化品种红黑树（red-black tree）实现的，原因会在下文介绍。

1.2 B树和B+树

目前大部分数据库系统及文件系统都采用B-Tree或其变种B+Tree作为索引结构，在本文的下一节会结合存储器原理及计算机存取原理讨论为什么B-Tree和B+Tree在被如此广泛用于索引，这一节先单纯从数据结构角度描述它们。

要理解B树，必须从二叉查找树（Binary search tree）讲起。



二叉查找树是一种查找效率非常高的数据结构，它有三个特点。

（1）每个节点最多只有两个子树。
（2）左子树都为小于父节点的值，右子树都为大于父节点的值。
（3）在n个节点中找到目标值，一般只需要log(n)次比较。

二叉查找树的结构不适合数据库，因为它的查找效率与层数相关。越处在下层的数据，就需要越多次比较。它的搜索时间复杂度为$O(log_2N)$，所以它的搜索效率和树的深度有关极端情况下，n个数据需要n次比较才能找到目标值。对于数据库来说，每进入一层，就要从硬盘读取一次数据，这非常致命，因为硬盘的读取时间远远大于数据处理时间，数据库读取硬盘的次数越少越好，这一点也会在后面深入剖析。

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。

1.2.1 B树

B树（B-tree）是一种树状数据结构，能够用来存储排序后的数据。这种数据结构能够让查找数据、循序存取、插入数据及删除的动作，都在对数时间内完成。B树，概括来说是一个一般化的二叉查找树，可以拥有多于2个子节点。与自平衡二叉查找树不同，B-树为系统最优化大块数据的读和写操作。B-tree算法减少定位记录时所经历的中间过程，从而加快存取速度。这种数据结

构常被应用在数据库和文件系统的实作上。

在B树中查找给定关键字的方法是，首先把根结点取来，在根结点所包含的关键字K1,…,Kn查找给定的关键字（可用顺序查找或二分查找法），若找到等于给定值的关键字，则查找成功；否则，一定可以确定要查找的关键字在Ki与Ki+1之间，Pi为指向子树根节点的指针，此时取指针Pi所指的结点继续查找，直至找到，或指针Pi为空时查找失败。

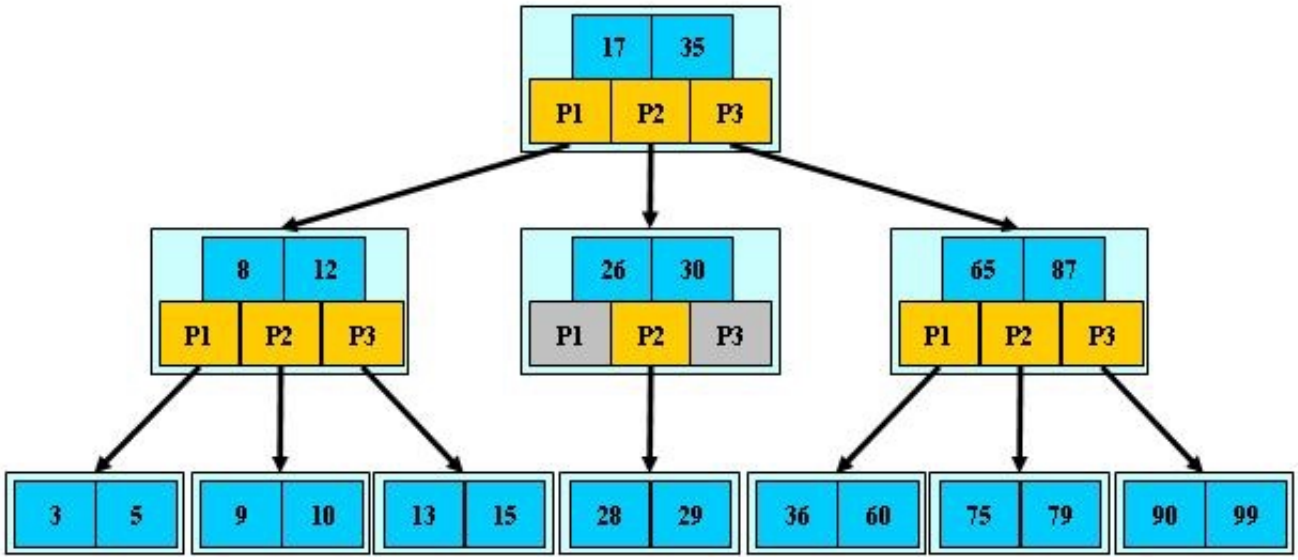
B树作为一种多路搜索树（并不是二叉的）：

- 定义任意非叶子结点最多只有M个儿子；且M>2；
- 根结点的儿子数为[2, M]；
- 除根结点以外的非叶子结点的儿子数为[M/2, M]；
- 每个结点存放至少M/2-1（取上整）和至多M-1个关键字；(至少2个关键字)
- 非叶子结点的关键字个数=指向儿子的指针个数-1；
- 非叶子结点的关键字：K[1], K[2], …, K[M-1]；且K[i] < K[i+1]；
- 非叶子结点的指针：P[1], P[2], …, P[M]；其中P[1]指向关键字小于K[1]的子树，P[M]指向关键字大于K[M-1]的子树，其它P[i]指向关键字属于(K[i-1], K[i])的子树；
- 所有叶子结点位于同一层；

此外，在B树中，除非数据已经填满，否则不会增加新的层。也就是说，B树追求层越少越好。

这样的数据结构，非常有利于减少读取硬盘的次数。假定一个节点可以容纳100个值，那么三层的B树可以容纳100万个数据，但若换成二叉查找树，则需要20层！假定操作系统一次读取一个节点，并且根节点保留在内存中，那么B树在100万个数据中查找目标值，只需要读取两次硬盘。

如下图为一个M=3的B树示例：



B树创建的示意图：

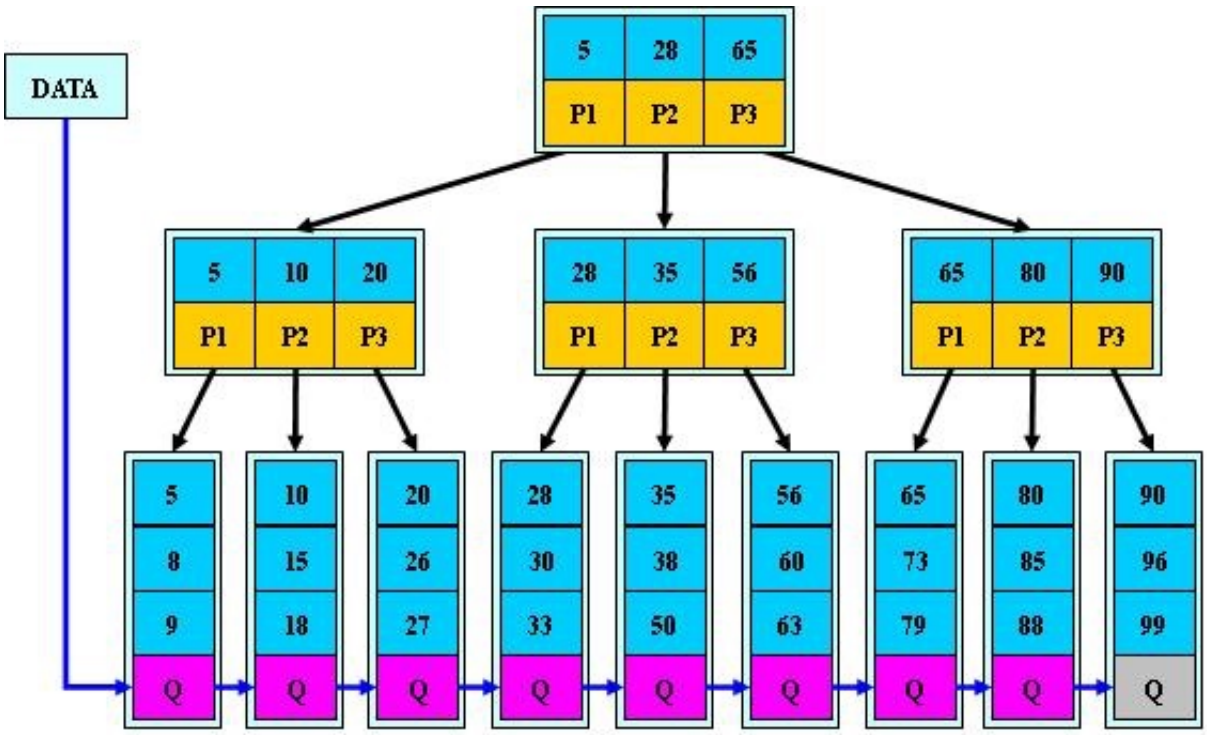
0006 0010

1.2.2 B+树

B+树是B树的变体，MySQL普遍使用B+Tree实现其索引结构。也是一种多路搜索树，其定义基本与B-树相同，除了：

- 1）非叶子结点的子树指针与关键字个数相同；
- 2）非叶子结点的子树指针P[i]，指向关键字值属于[K[i], K[i+1])的子树（B-树是开区间）；
- 3）为所有叶子结点增加一个链指针；
- 4）所有关键字都在叶子结点出现；

下图为M=3的B+树的示意图：



B+树的搜索与B树也基本相同，区别是B+树只有达到叶子结点才命中（B树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；

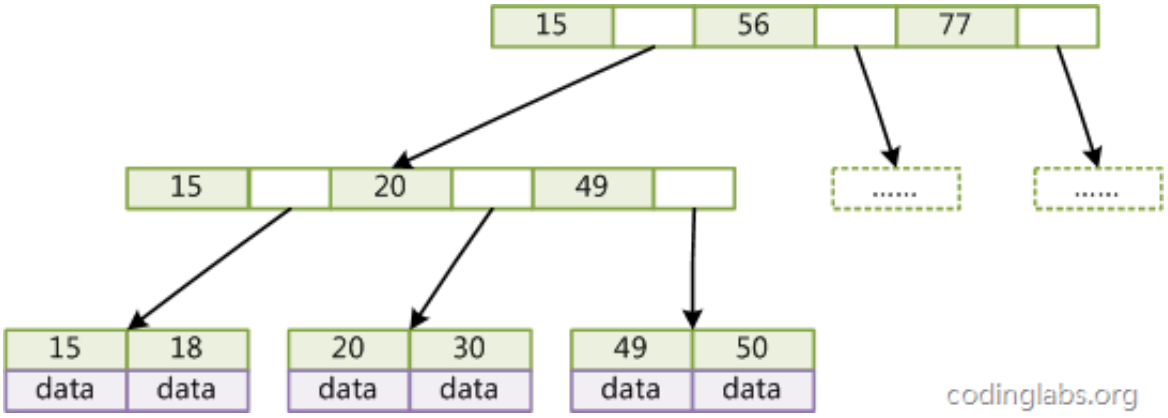
B+树的性质：

- 1.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；
- 2.不可能在非叶子结点命中；

- 3.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；
- 4.更适合文件索引系统。

下面为一个B+树创建的示意图：

一般在数据库系统或文件系统中使用的B+ Tree结构都在经典B+ Tree的基础上进行了优化，增加了顺序访问指针。做这个优化的目的是为了提高区间访问的性能，例如下图中如果要查询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，极大提到了区间查询效率。



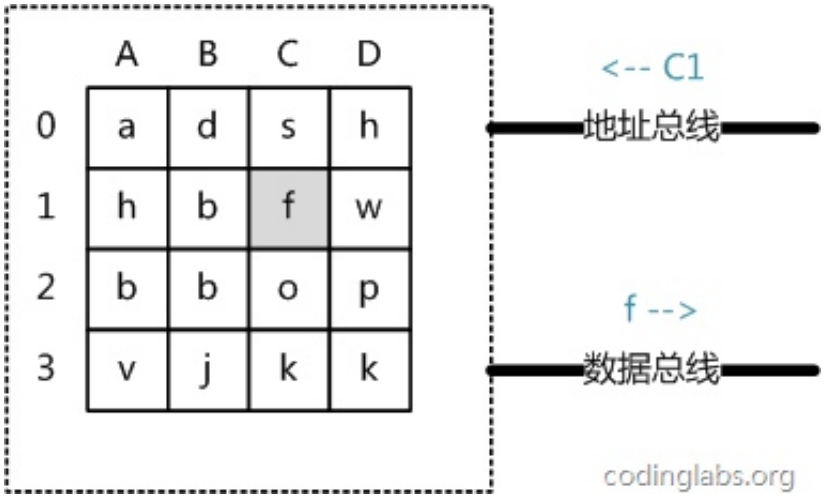
二、计算机组成原理

上文说过，红黑树等数据结构也可以用来实现索引，但是文件系统及数据库系统普遍采用B-/+Tree作为索引结构，这一节将结合计算机组成原理相关知识讨论B-/+Tree作为索引的理论基础。

一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗，相对于内存存取，I/O存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘I/O操作次数的渐进复杂度。换句话说，索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数。下面先介绍内存和磁盘存取原理，然后再结合这些原理分析B-/+Tree作为索引的效率。

2.1 主存存取原理

目前计算机使用的主存基本都是随机读写存储器（RAM），现代RAM的结构和存取原理比较复杂，这里本文抛却具体差别，抽象出一个十分简单的存取模型来说明RAM的工作原理。



从抽象角度看，主存是一系列的存储单元组成的矩阵，每个存储单元存储固定大小的数据。每个存储单元有唯一的地址，现代主存的编址规则比较复杂，这里将其简化成一个二维地址：通过一个行地址和一个列地址可以唯一定位到一个存储单元。上图展示了一个4 x 4的主存模型。

主存的存取过程如下：

当系统需要读取主存时，则将地址信号放到地址总线上传给主存，主存读到地址信号后，解析信号并定位到指定存储单元，然后将此存储单元数据放到数据总线上，供其它部件读取。写主存的过程类似，系统将要写入单元地址和数据分别放在地址总线和数据总线上，主存读取两个总线的内容，做相应的写操作。

这里可以看出，主存存取的时间仅与存取次数呈线性关系，因为不存在机械操作，两次存取的数据的“距离”不会对时间有任何影响，例如，先取A0再取A1和先取A0再取D3的时间消耗是一样的。

2.2 磁盘存取原理

上文说过，索引一般以文件形式存储在磁盘上，索引检索需要磁盘I/O操作。与主存不同，磁盘I/O存在机械运动耗费，因此磁盘I/O的时间消耗是巨大的。

磁盘读取数据靠的是机械运动，当需要从磁盘读取数据时，系统会将数据逻辑地址传给磁盘，磁盘的控制电路按照寻址逻辑将逻辑地址翻译成物理地址，即确定要读的数据在哪个磁道，哪个扇区。

为了读取这个扇区的数据，需要将磁头放到这个扇区上方，为了实现这一点，磁头需要移动对准相应磁道，这个过程叫做寻道，所耗费时间叫做寻道时间，然后磁盘旋转将目标扇区旋转到磁头下，这个过程耗费的时间叫做旋转时间，最后便是对读取数据的传输。 所以每次读取数据花费的时间可以分为寻道时间、旋转延迟、传输时间三个部分。其中：

- 寻道时间是磁臂移动到指定磁道所需要的时间，主流磁盘一般在5ms以下。
- 旋转延迟就是我们经常听说的磁盘转速，比如一个磁盘7200转，表示每分钟能转7200次，也就是说1秒钟能转120次，旋转延迟就是1/120/2 = 4.17ms。
- 传输时间指的是从磁盘读出或将数据写入磁盘的时间，一般在零点几毫秒，相对于前两个时间可以忽略不计。

那么访问一次磁盘的时间，即一次磁盘IO的时间约等于5+4.17 = 9ms左右，听起来还挺不错的，但要知道一台500 -MIPS的机器每秒可以执行5亿条指令，因为指令依靠的是电的性质，换句话说执行一次IO的时间可以执行40万条指令，数据库动辄十万百万乃至千万级数据，每次9毫秒的时间，显然是个灾难。

2.3 局部性原理与磁盘预读

由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分分之一，因此为了提高效率，要尽量减少磁盘I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的局部性原理：当一个数据被用到时，其附近的数据也通常会马上被使用。程序运行期间所需要的数据通常比较集中。

由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高I/O效率。预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为4k），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

三、B-/+Tree索引的性能分析

到这里终于可以分析为何数据库索引采用B-/+Tree存储结构了。上文说过数据库索引是存储到磁盘的而我们又一般以使用磁盘I/O次数来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-1个节点（根节点常驻内存）。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$O(h) = O(log_d N)$。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，如果我们采用B-Tree存储结构，搜索时I/O次数一般不会超过3次，所以用B-Tree作为索引结构效率是非常高的。

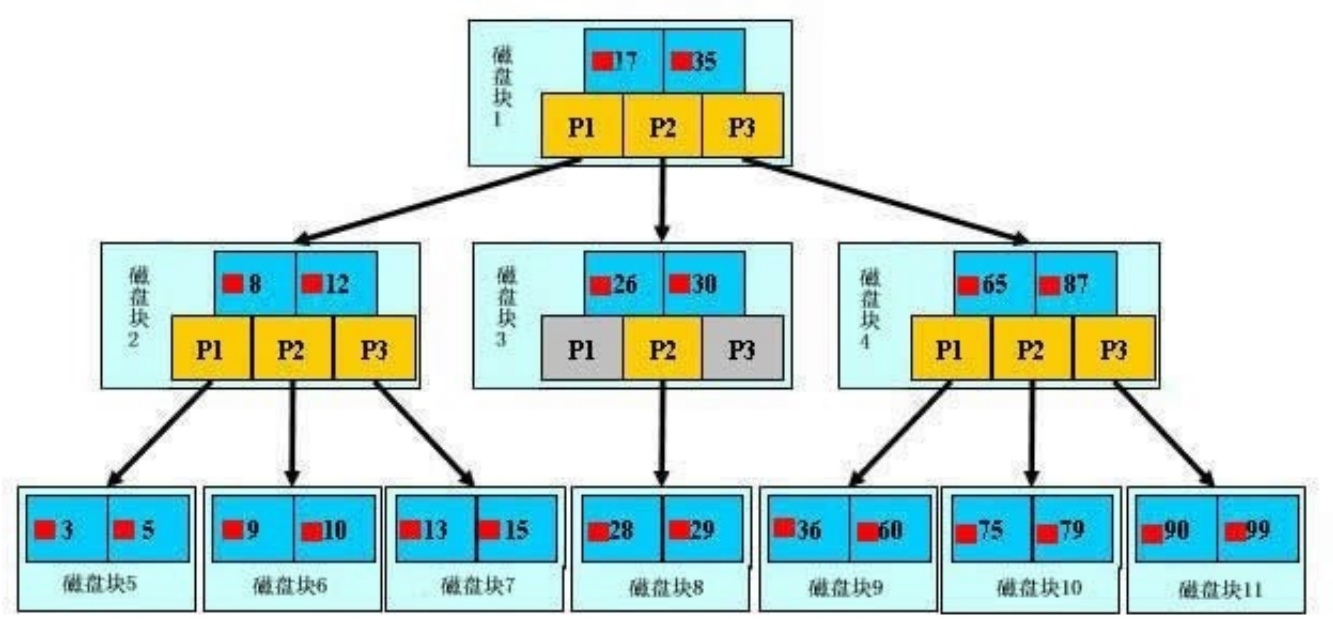
3.1 B+树性能分析

从上面介绍我们知道，B树的搜索复杂度为O(h)=O(logdN)，所以树的出度d越大，深度h就越小，I/O的次数就越少。B+Tree恰恰可以增加出度d的宽度，因为每个节点大小为一个页大小，所以出度的上限取决于节点内key和data的大小：

```
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，从而拥有更好的性能。

3.2 B+树查找过程



B-树和B+树查找过程基本一致。如上图所示，如果要查找数据项29，那么首先会把磁盘块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，内存时间因为非常短（相比磁盘的IO）可以忽略不计，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，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。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，每个数据项都要发生一次IO，那么总共需要百万次的IO，显然成本非常非常高。

这一章从理论角度讨论了与索引相关的数据结构与算法问题，下一章将讨论B+Tree是如何具体实现为MySQL中索引，同时将结合MyISAM和InnDB存储引擎介绍非聚集索引和聚集索引两种不同

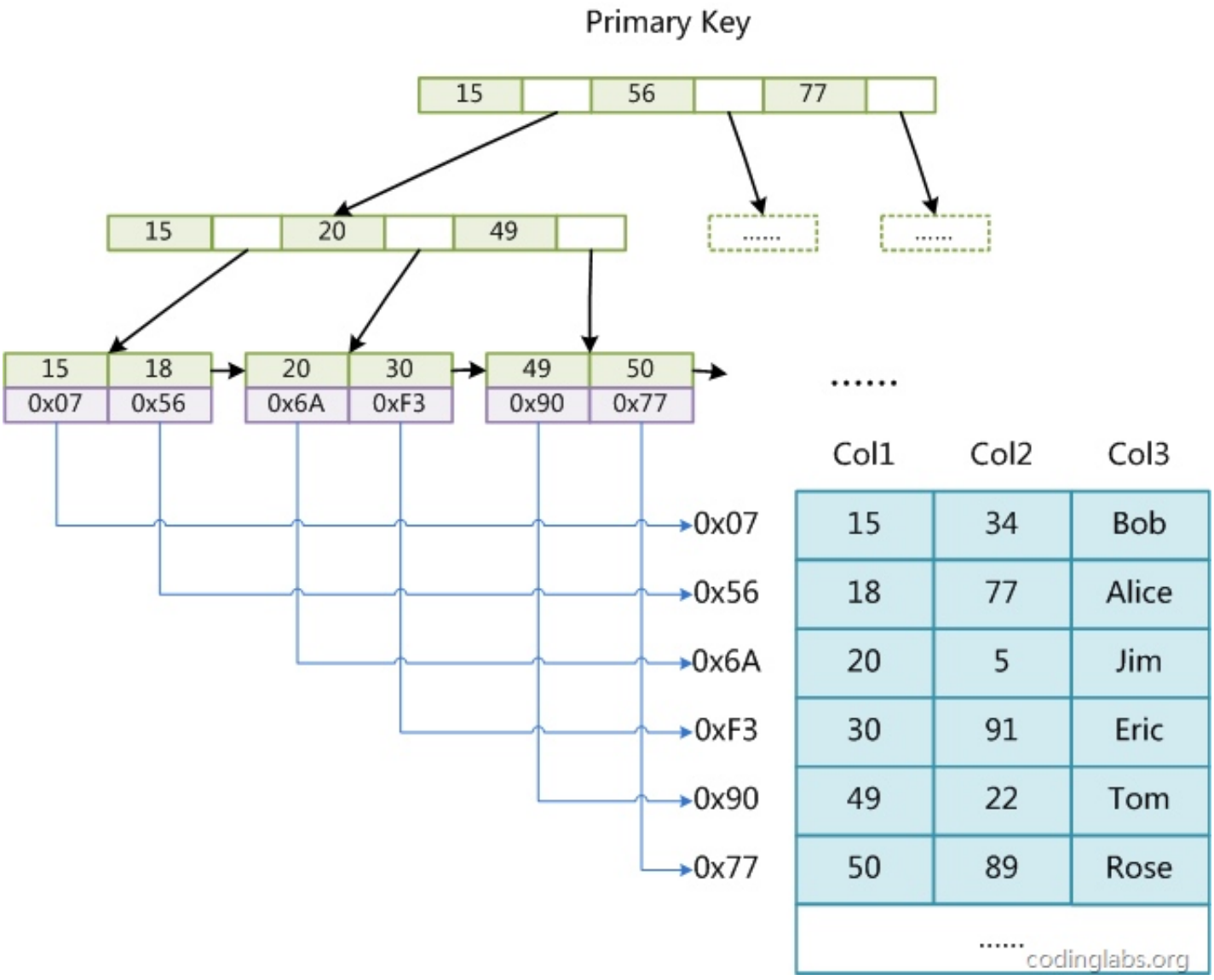
的索引实现形式。

四、MySQL索引实现

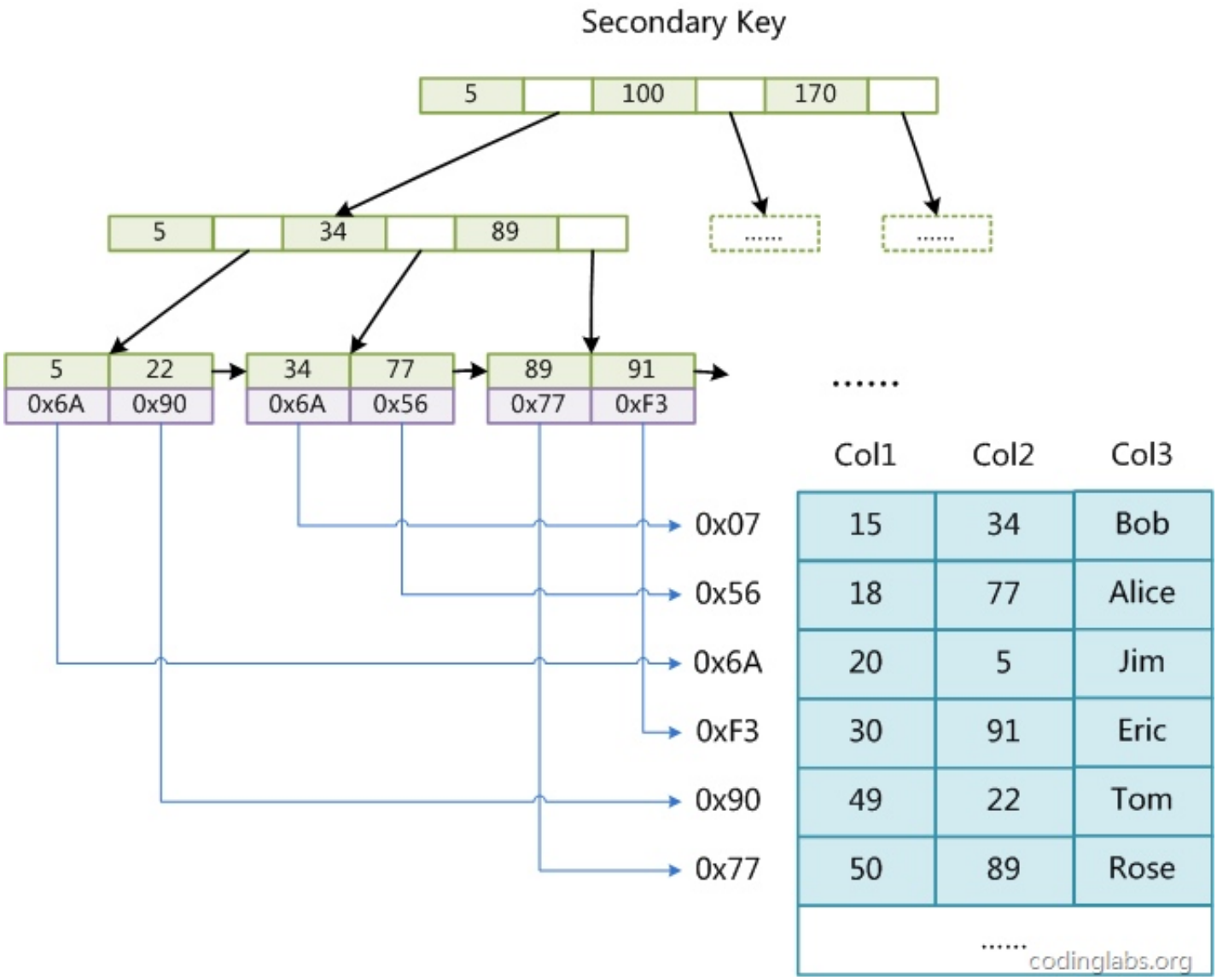
在MySQL中，索引属于存储引擎级别的概念，不同存储引擎对索引的实现方式是不同的，本文主要讨论MyISAM和InnoDB两个存储引擎的索引实现方式。

4.1 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。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，则上图是一个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：



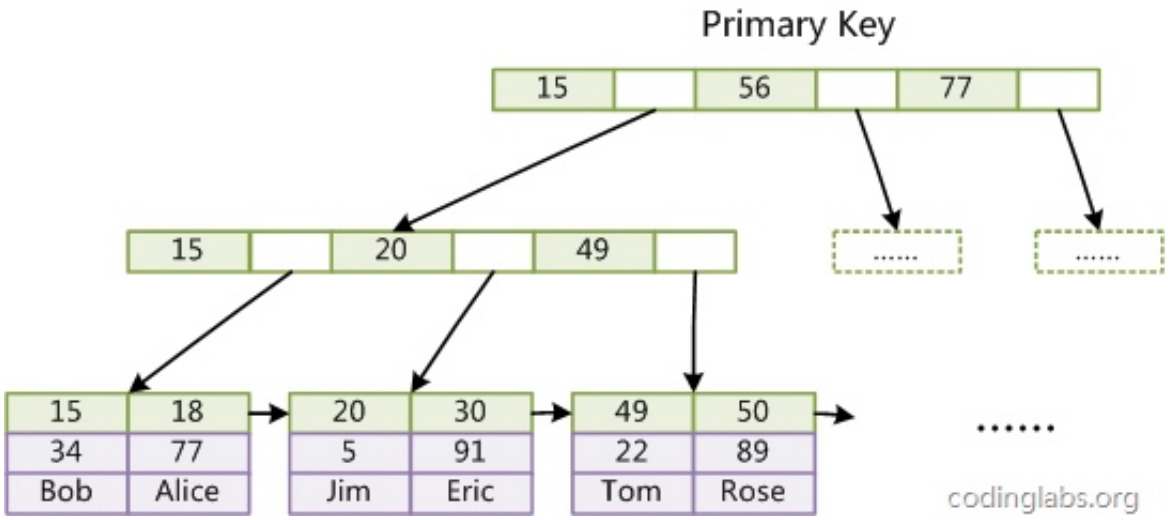
同样也是一颗B+Tree，data域保存数据记录的地址。因此，MyISAM中索引检索的算法为首先按照B+Tree搜索算法搜索索引，如果指定的Key存在，则取出其data域的值，然后以data域的值为地址，读取相应数据记录。

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。

4.2 InnoDB索引实现

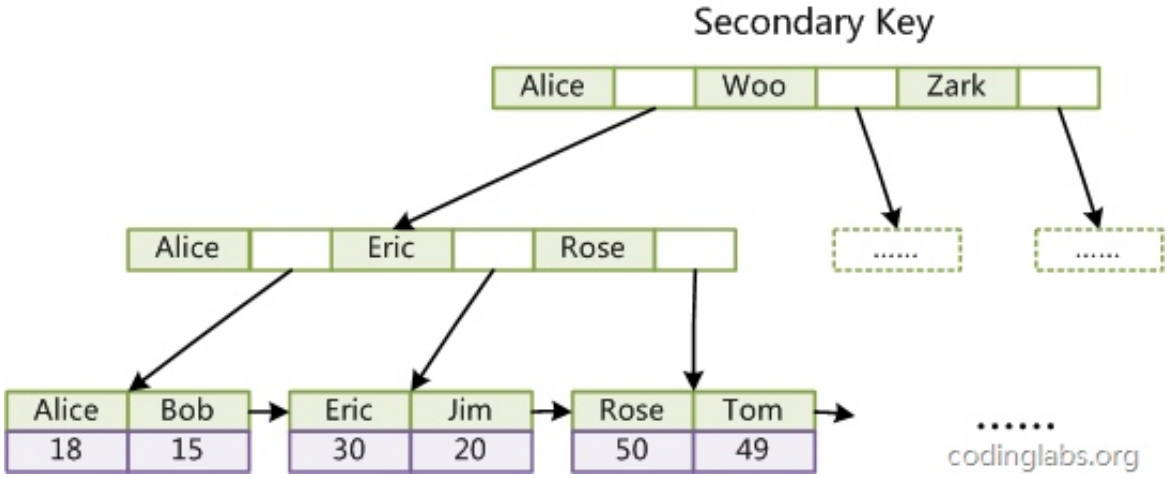
虽然InnoDB也使用B+Tree作为索引结构，但具体实现方式却与MyISAM截然不同。

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。



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。

五、索引使用策略及优化

MySQL的优化主要分为结构优化（Scheme optimization）和查询优化（Query optimization）。本章讨论的高性能索引策略主要属于结构优化范畴。本章的内容完全基于上文的理论基础，实际上一旦理解了索引背后的机制，那么选择高性能的策略就变成了纯粹的推理，并且可以理解这些策略背后的逻辑。

5.1 联合索引及最左前缀原理

> 联合索引（复合索引）

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 |
| 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。

> 最左前缀原理

我们再来详细介绍一下联合索引的查询。还是上面例子，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

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

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

比如：

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

> 前缀索引

除了联合索引之外，对mysql来说其实还有一种前缀索引。前缀索引就是用列的前缀代替整个列作为索引key，当前缀长度合适时，可以做到既使得前缀索引的选择性接近全列索引，同时因为索引key变短而减少了索引文件的大小和维护开销。

- 字符串列(varchar,char,text等)，需要进行全字段匹配或者前匹配。也就是='xxx' 或者 like 'xxx%'
- 字符串本身可能比较长，而且前几个字符就开始不相同。比如我们对中国人的姓名使用前缀索引就没啥意义，因为中国人名字都很短，另外对收件地址使用前缀索引也不是很实用，因为一方面收件地址一般都是以XX省开头，也就是说前几个字符都是差不多的，而且收件地址进行检索一般都是like '%xxx%'，不会用到前匹配。相反对外国人的姓名可以使用前缀索引，因为其字符较长，而且前几个字符的选择性比较高。同样电子邮件也是一个可以使用前缀索引的字段。
- 前一半字符的索引选择性就已经接近于全字段的索引选择性。如果整个字段的长度为20，索引选择性为0.9，而我们对前10个字符建立前缀索引其选择性也只有0.5，那么我们需要继续加大前缀字符的长度，但是这个时候前缀索引的优势已经不明显，没有太大的建前缀索引的必要了。

一些文章中也提到：

MySQL 前缀索引能有效减小索引文件的大小，提高索引的速度。但是前缀索引也有它的坏处：

MySQL 不能在 ORDER BY 或 GROUP BY 中使用前缀索引，也不能把它们用作覆盖索引 (Covering Index)。

5.2 索引优化策略

- 最左前缀匹配原则，上面讲到了
- 主键外检一定要建索引
- 对 where,on,group by,order by 中出现的列使用索引
- 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，而一些状态、性别字段可能在大数据面前区分度就是0
- 对较小的数据列使用索引,这样会使索引文件更小,同时内存中也可以装载更多的索引键
- 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。所以语句应该写成create_time = unix_timestamp(’2014-05-29’);
- 为较长的字符串使用前缀索引
- 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
- 不要过多创建索引, 权衡索引个数与DML之间关系，DML也就是插入、删除数据操作。这里需要权衡一个问题，建立索引的目的是为了提高查询效率的，但建立的索引过多，会影响插入、删除数据的速度，因为我们修改的表数据，索引也需要进行调整重建
- 对于like查询，”%”不要放在前面。

```
SELECT * FROMhoudunwangWHEREunameLIKE'后盾%' -- 走索引
SELECT * FROMhoudunwangWHEREunameLIKE "%后盾%" -- 不走索引
```

- 查询where条件数据类型不匹配也无法使用索引
- 字符串与数字比较不使用索引;

```
CREATE TABLEa(achar(10));
EXPLAIN SELECT * FROMaWHEREa="1" - 走索引
EXPLAIN SELECT * FROM a WHERE a=1 - 不走索引
```

正则表达式不使用索引,这应该很好理解,所以为什么在SQL中很难看到regexp关键字的原因