MySQL索引背后数据结构及算法原理

摘要

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。

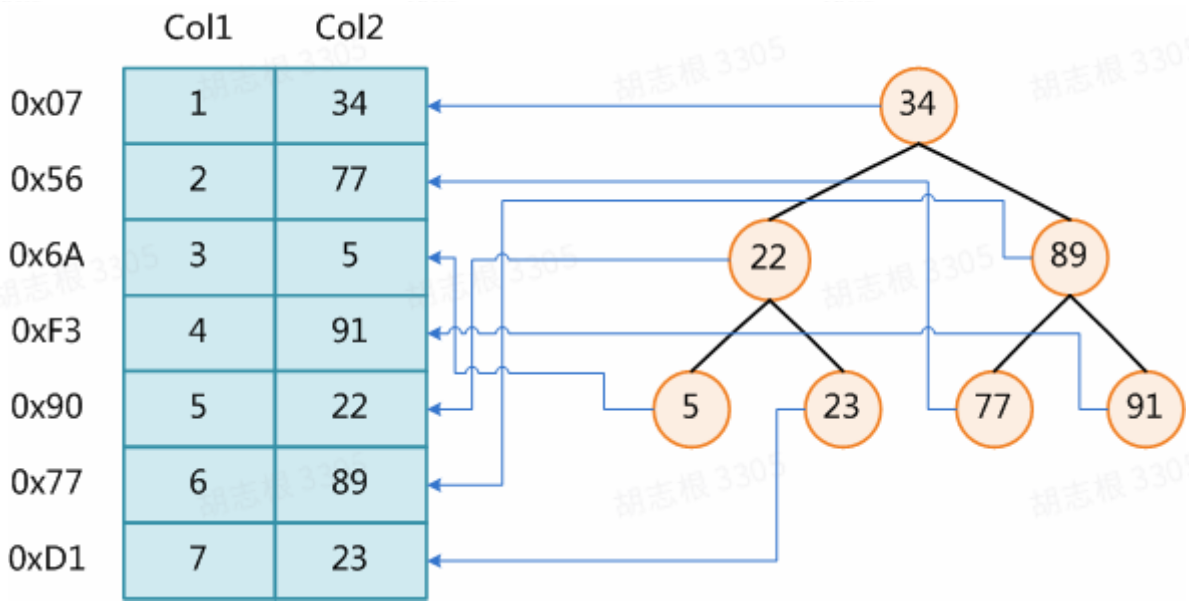
数据结构及算法基础

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：



上图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(logN)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。

B-Tree和B+Tree

提个问题：**『B-树』 应该读『B树』还是『B减树』？**

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。

B-Tree

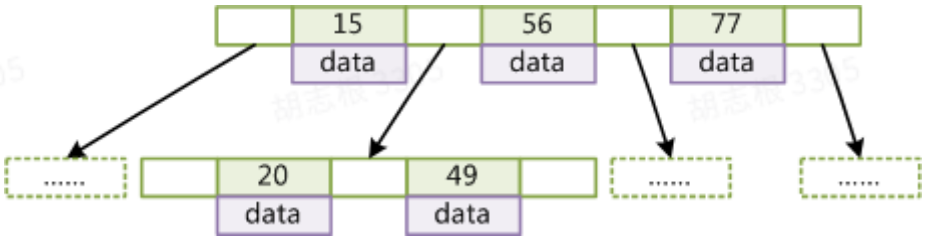
维基百科的定义https://zh.wikipedia.org/wiki/B%E6%A0%91太过专业，开始看的时候有点难懂，感觉很懵逼的，下面使用“人话”来说明下。

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。那么B-Tree是满足下列条件的数据结构：

1. d为大于1的一个正整数，称为B-Tree的度或者分支因子；
2. h为一个正整数，称为B-Tree的高度；
3. 每个非叶子节点由n-1个key和n个指针组成，其中d<=n<=2d；

4. 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
5. 所有叶节点具有相同的深度，等于树高h。
6. key和指针互相间隔，节点两端是指针。
7. 一个节点中的key从左到右非递减排列。
8. 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。
9. 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a，其中a为node的第一个key的值。
10. 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b，其中b为node的最后一个key的值。
11. 如果某个指针在节点node的左右相邻key分别是a和b且不为null，则其指向节点的所有key小于b且大于a。

下图是一个d=2的B-Tree示意图。



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：

```
BTree_Search(node, key)
{
    if(node == null) return null;
    foreach(node.key)
    {
        if(node.key[i] == key) return node.data[i];
            if(node.key[i] > key) return BTree_Search(point[i]->node);
    }
    return BTree_Search(point[i+1]->node);
}
data = BTree_Search(root, my_key);
```

检索一个key，其查找节点个数的渐进复杂度为O(log_dN)。从这点可以看出，B-Tree是一个非常有效率的索引数据结构。

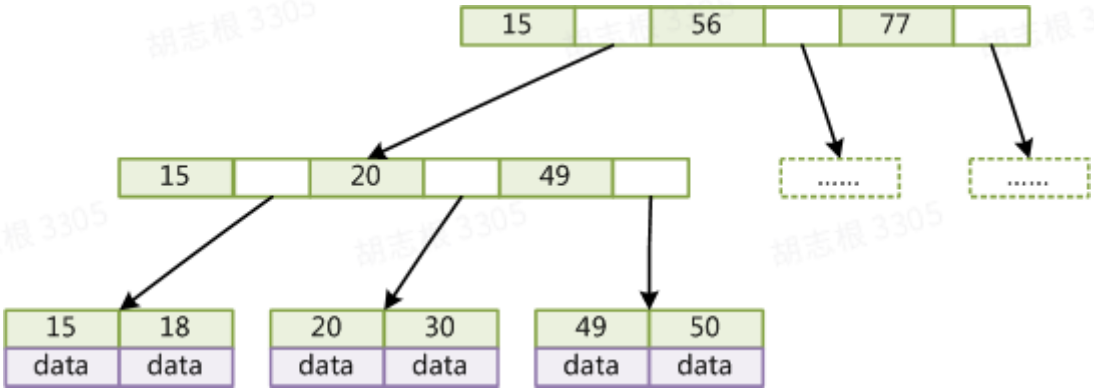
另外，由于插入删除新的数据记录会破坏B-Tree的性质，因此在插入删除时，需要对树进行一个分裂、合并、转移等操作以保持B-Tree性质，本文不打算完整讨论B-Tree这些内容，因为已经有许多资料详细说明了B-Tree的数学性质及插入删除算法，有兴趣的同学可以自行查阅资料学习。

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：

1. 每个节点的指针上限为2d - 1而不是2d，即消除最左边的指针；
2. 内节点不存储data，只存储key；叶子节点不存储指针；只有叶子节点才存储data；

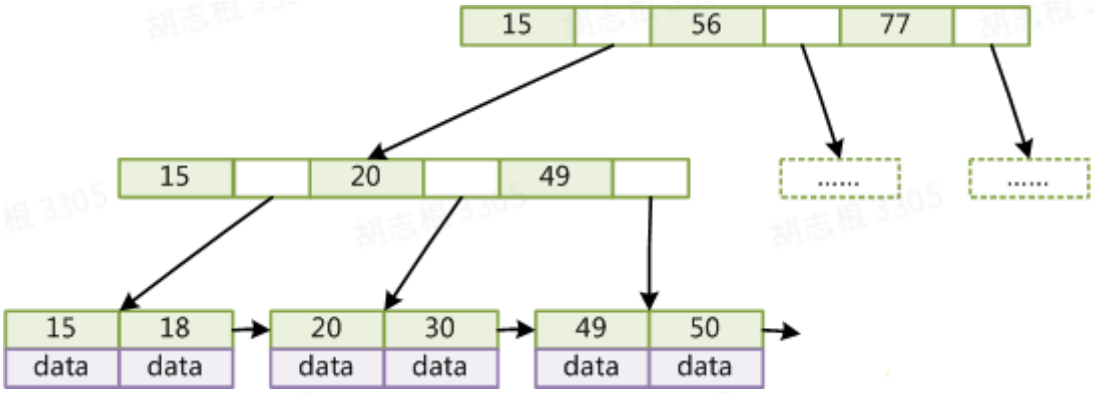
下图是一个简单的B+Tree示意



由于并不是所有节点都具有相同的域，因此B+Tree中叶节点和内节点一般大小不同。这点与B-Tree不同，虽然B-Tree中不同节点存放的key和指针可能数量不一致，但是每个节点的域的上限是一致的，所以在实现中B-Tree往往对每个节点申请同等大小的空间。
一般来说，B+Tree比B-Tree更适合实现外存储索引结构，具体原因与外存储器原理及计算机存取原理有关，将在下面讨论。

带有顺序访问指针的B+Tree

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。



如上图所示，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，例如图中如果要查询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。

这一节对B-Tree和B+Tree进行了一个简单的介绍，下一节结合存储器存取原理介绍为什么目前B+Tree是数据库系统实现索引的首选数据结构。

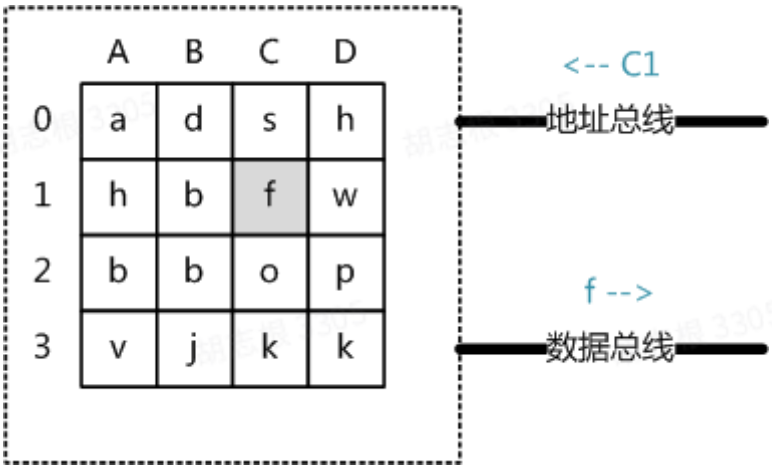
为什么使用B-Tree（B+Tree）

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。

主存存取原理

目前计算机使用的主存(Memory)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：

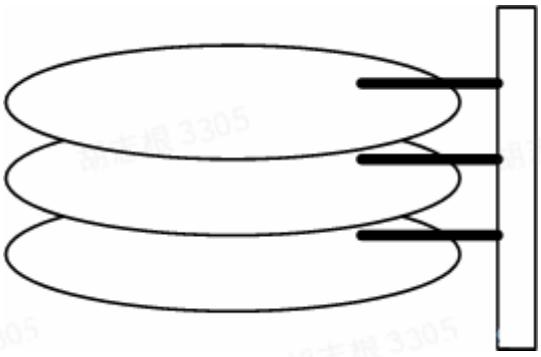
- 当系统需要读取主存时，则将地址信号放到地址总线上传给主存，主存读到地址信号后，解析信号并定位到指定存储单元，然后将此存储单元数据放到数据总线上，供其它部件读取。
- 写主存的过程类似，系统将要写入单元地址和数据分别放在地址总线和数据总线上，主存读取两个总线的内容，做相应的写操作。

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。

磁盘存取原理

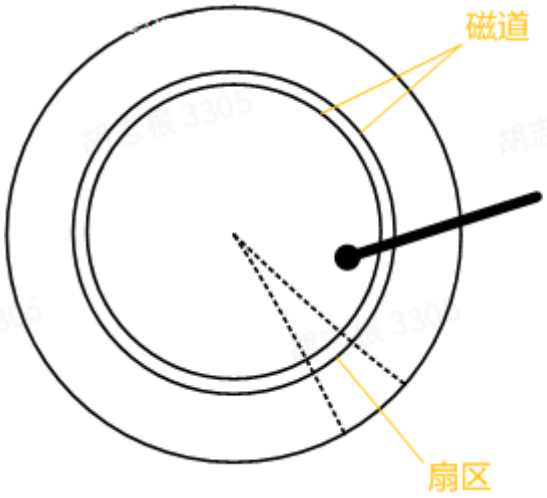
上文说过，索引一般以文件形式存储在磁盘(Disk)上，索引检索需要磁盘I/O操作。与主存不同，磁盘I/O存在机械运动耗费，因此磁盘I/O的时间消耗是巨大的。

下图是经典磁盘的整体结构示意图



一个磁盘由大小相同且同轴的圆形**盘片**组成，磁盘可以转动（各个磁盘必须同步转动）。在磁盘的一侧有**磁头支架**，磁头支架固定了一组**磁头**，每个磁头负责存取一个磁盘的内容。磁头不能转动，但是可以沿磁盘半径方向运动（实际是斜切向运动），每个磁头同一时刻也必须是同轴的，即从正上方向下看，所有磁头任何时候都是重叠的（不过目前已经有多磁头独立技术，可不受此限制）。

下图是磁盘结构的示意图



盘片被划分成一系列同心环，圆心是盘片中心，每个同心环叫做一个磁道，所有半径相同的磁道组成一个柱面。磁道被沿半径线划分成一个个小的段，每个段叫做一个扇区，每个扇区是磁盘的最小存储单元。

为了简单起见，我们下面假设磁盘只有一个盘片和一个磁头。
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。

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(中断信号)，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。

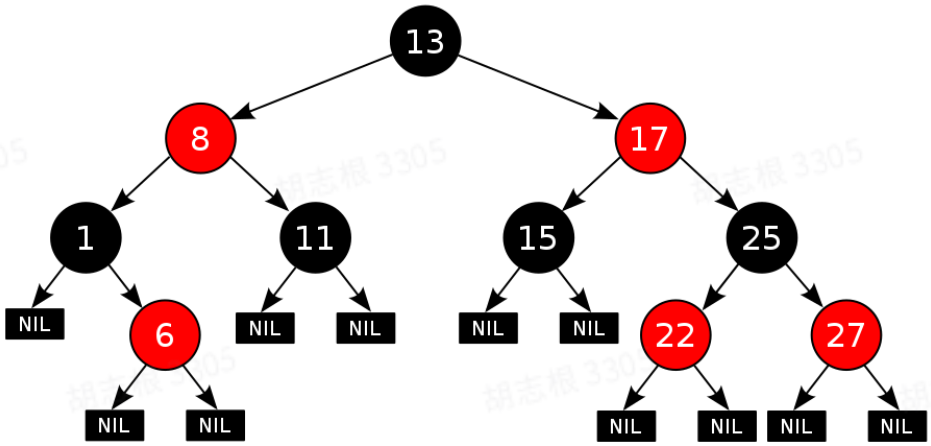
B-/+Tree索引的性能分析

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(高度)个节点。MySQL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。

在具体实现的时候，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https://zh.wikipedia.org/wiki/%E7%BA%A2%E9%BB%91%E6%A0%91这种结构(实质就是特特化的二叉平衡树)，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。

上文还说过，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：

```
d_{max}=floor(pagesize / (keysize + datasize + pointsize))
```

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。

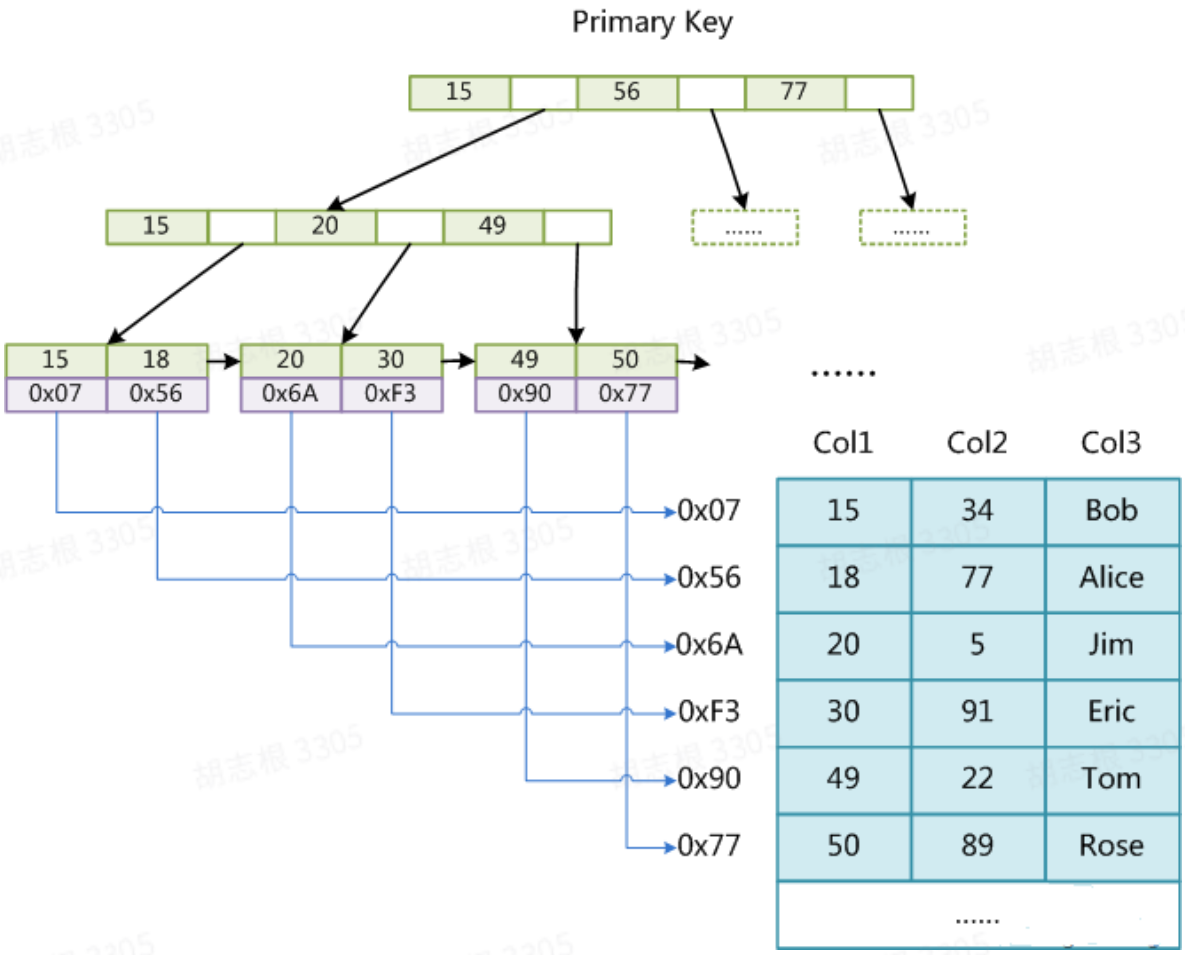
这节从理论角度讨论了与索引相关的数据结构与算法问题，下面将讨论B+Tree是如何具体实现为MySQL中索引，同时将结合MyISAM和InnDB存储引擎介绍非聚集索引和聚集索引两种不同的索引实现形式。

MySQL索引实现

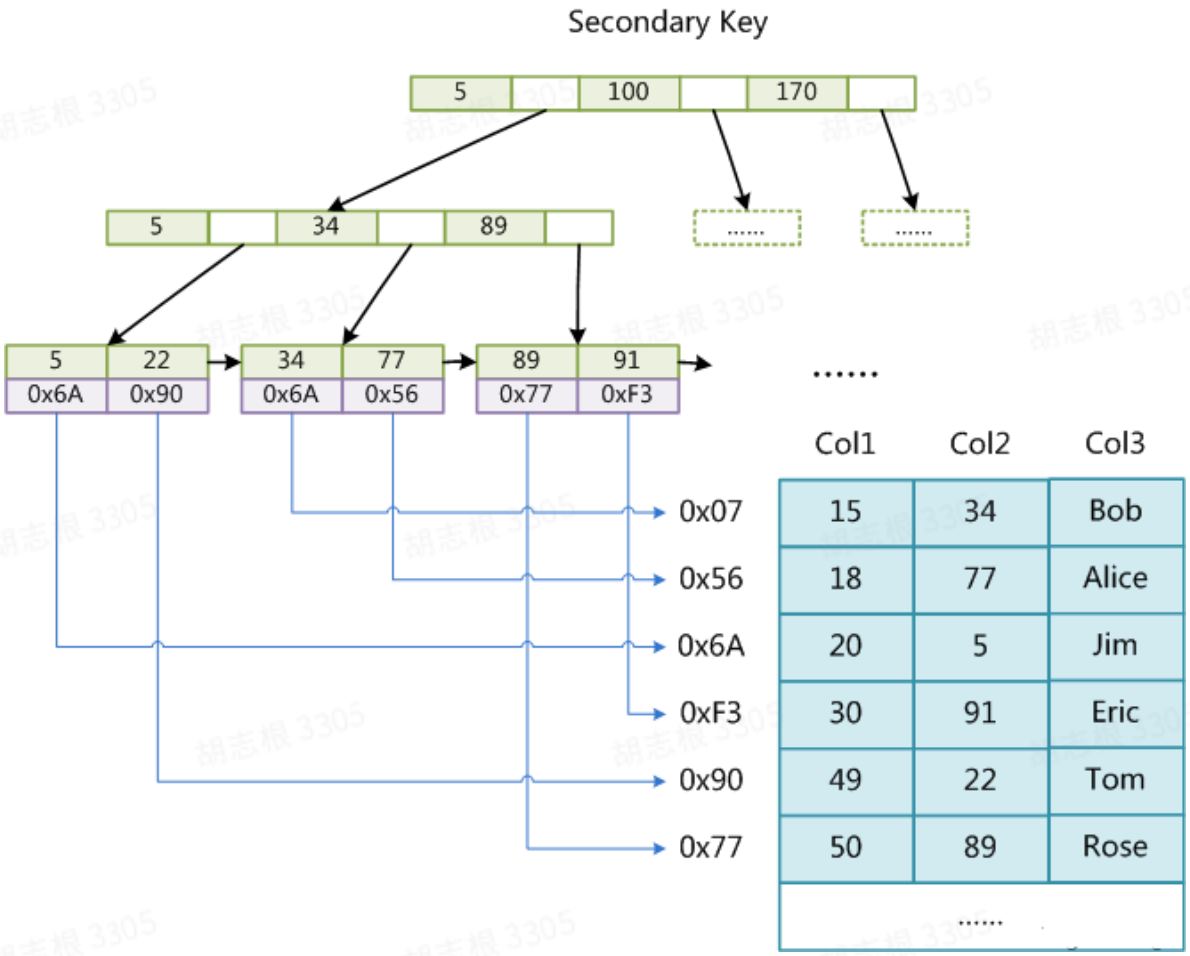
在MySQL中，索引属于存储引擎级别的概念，不同存储引擎对索引的实现方式是不同的，本文主要讨论MyISAM和InnoDB两个存储引擎的索引实现方式。

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：



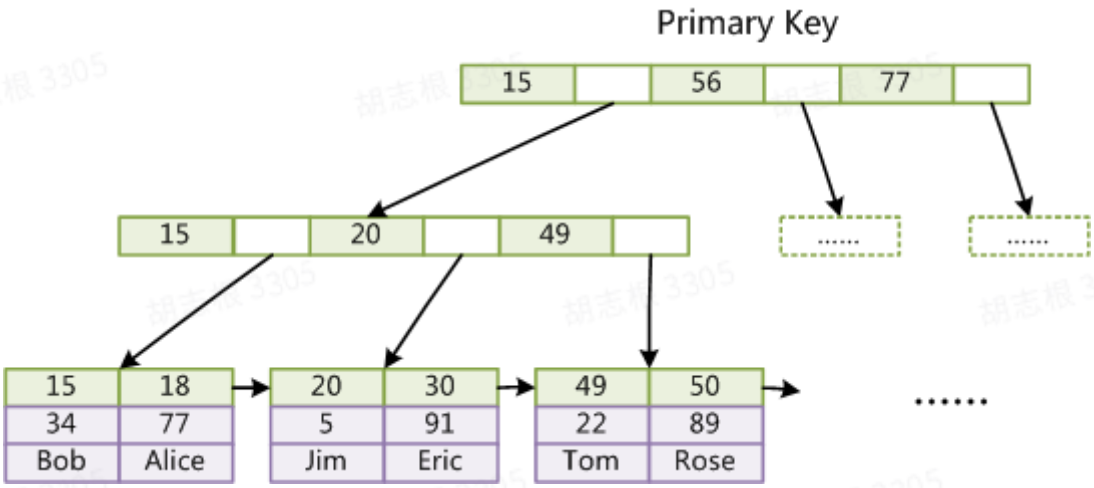
这里设表一共有三列，假设我们以Col1为主键，则图8是一个MyISAM表的主索引（Primary key）示意。可以看出MyISAM的索引文件仅仅保存数据记录的地址。在MyISAM中，主索引和辅助索引（Secondary key）在结构上没有任何区别，只是主索引要求key是唯一的，而辅助索引的key可以重复。如果我们在Col2上建立一个辅助索引，则此索引的结构如下图所示：



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。

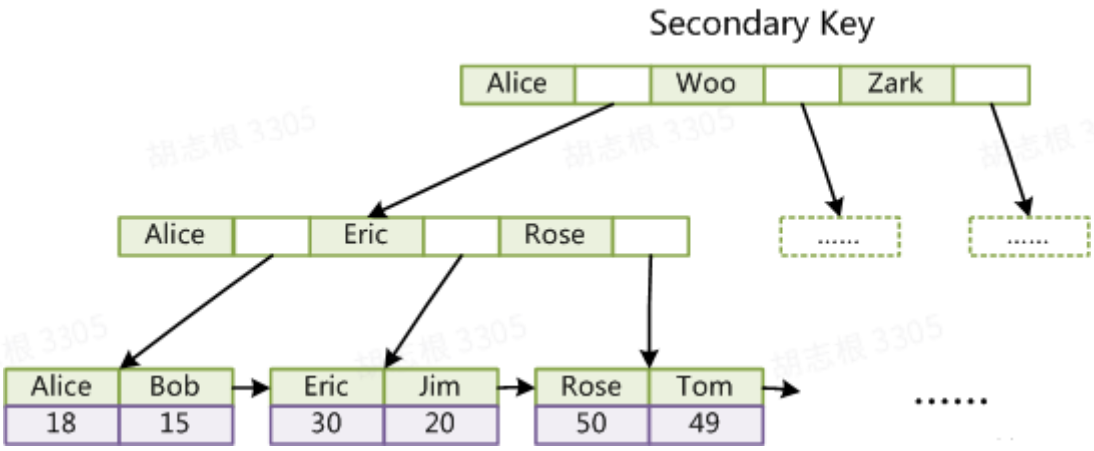
InnoDB索引实现

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，为定义在Col3上的一个辅助索引：



可以看出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

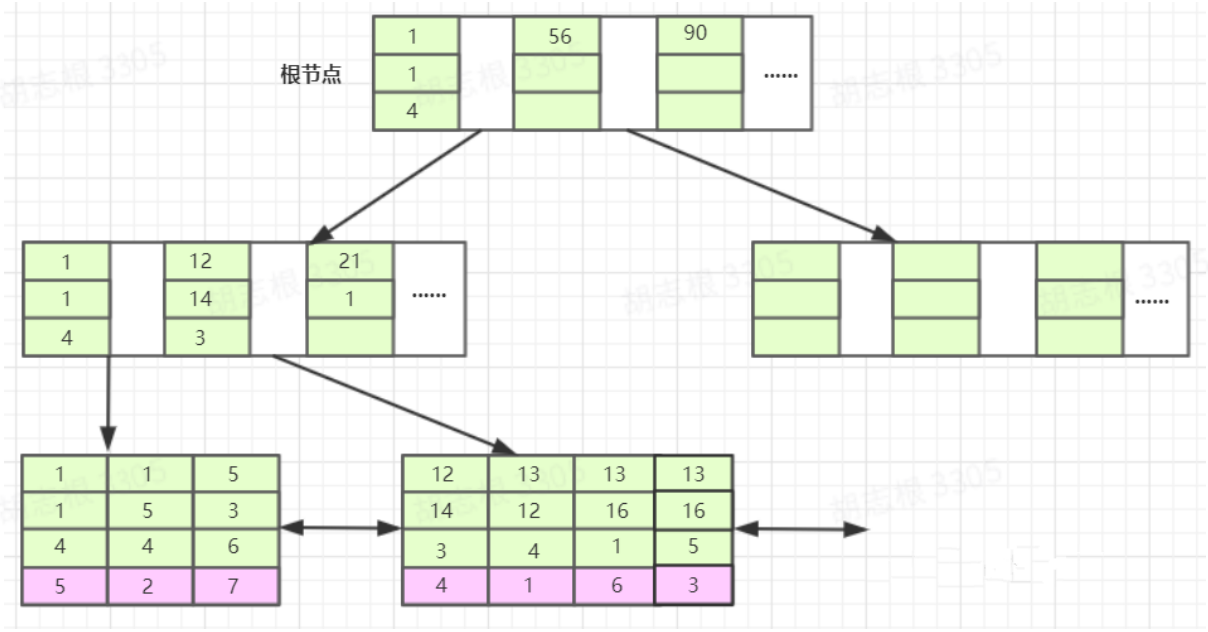
- 例如知道了InnoDB的索引实现后，就很容易明白为什么不建议使用过长的字段作为主键，因为所有辅助索引都引用主索引，过长的主索引会令辅助索引变得过大。
- 再例如，用非单调的字段作为主键在InnoDB中不是个好主意，因为InnoDB数据文件本身是一颗B+Tree，非单调的主键会造成在插入新记录时数据文件为了维持B+Tree的特性而频繁的分裂调

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。

联合索引(InnoDB)

实际工程应用中，多列联合索引使用也比较多，这里特别说明下存储结构以及相关的原理

```
table t1 (a int, b int , c int, d int, e varchar(100))
create index idx_t1_bcd on t1(b,c,d);
```



a	b	c	d	e
1	13	12	4	dll
2	1	5	4	doc
3	13	16	5	img
4	12	14	3	xml
5	1	1	4	txt
6	13	16	1	exe
7	5	3	6	pdf
.....				

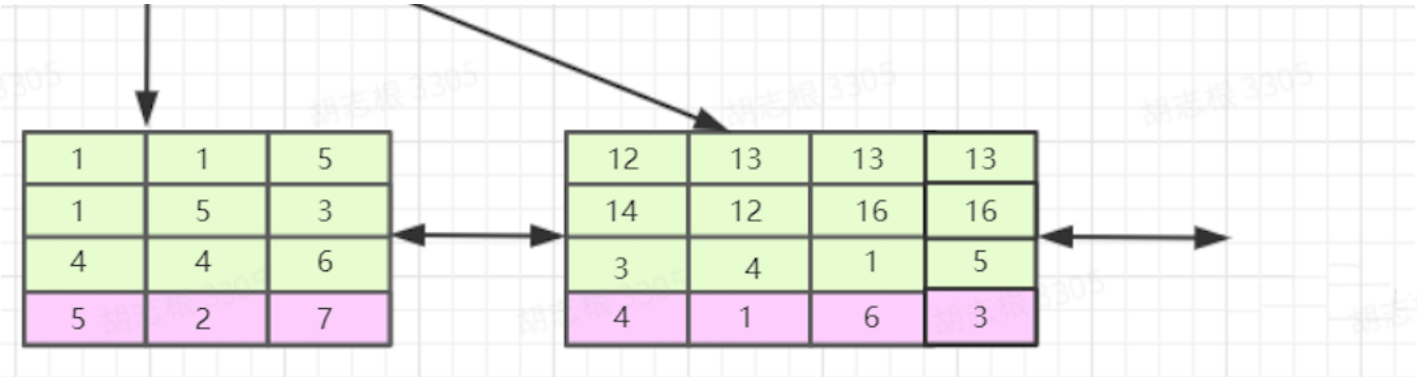
看T1表，InnoDB会使用主键索引在B+树维护索引和数据文件，然后我们创建了一个联合索引（b，c，d）也会生成一个索引树，同样是B+树的结构，只不过它的data部分存储的是联合索引所在行的主键值（上图叶子节点紫色背景部分）

对于联合索引来说只不过比单值索引多了几列，而这些索引列全都出现在索引树上。对于联合索引，存储引擎会首先根据第一个索引列排序，如上图我们可以单看第一个索引列，如，1 1 5 12 13....他是单调递增的；如果第一列相等则再根据第二列排序，依次类推就构成了上图的索引树，上图中的1 1 4，1 1 5以及13 12 4,13 16 1,13 16 5就可以说明这种情况。

最左前缀匹配原则

之所以会有最左前缀匹配原则和联合索引的索引构建方式及存储结构是有关系的。

联合索引是首先使用多列索引的第一列构建的索引树，用上面idx_t1_bcd(b,c,d)的例子就是优先使用b列构建，当b列值相等时再以c列排序，若c列的值也相等则以d列排序。我们可以取出索引树的叶子节点看一下。



索引的第一列也就是b列可以说是从左到右单调递增的，但我们看c列和d列并没有这个特性，它们只能在b列值相等的情况下这个小范围内递增，如第一叶子节点的第1、2个元素和第二个叶子节点的后三个元素。

由于联合索引是上述那样的索引构建方式及存储结构，所以联合索引只能从多列索引的第一列开始查找。所以如果你的查找条件不包含b列如（c,d）、(c）、(d)是无法应用索引的，以及跨列也是无法完全用到索引如(b,d)，只会用到b列索引。

这就像我们的电话本一样，有名和姓以及电话，名和姓就是联合索引。在姓可以以姓的首字母排序，姓的首字母相同的情况下，再以名的首字母排序。
如：

```
1 M
```

```
    毛 不易    178********
    马 化腾    183********
    马 云      188********
Z
    张 杰      189********
    张 靓颖    138********
    张 艺兴    176********
```

我们知道名和姓是很快就能够从姓的首字母索引定位到姓，然后定位到名，进而找到电话号码，因为所有的姓从上到下按照既定的规则（首字母排序）是有序的，而名是在姓的首字母一定的条件下也是按照名的首字母排序的，但是整体来看，所有的名放在一起是无序的，所以如果只知道名查找起来就比较慢，因为无法用已排好的结构快速查找。

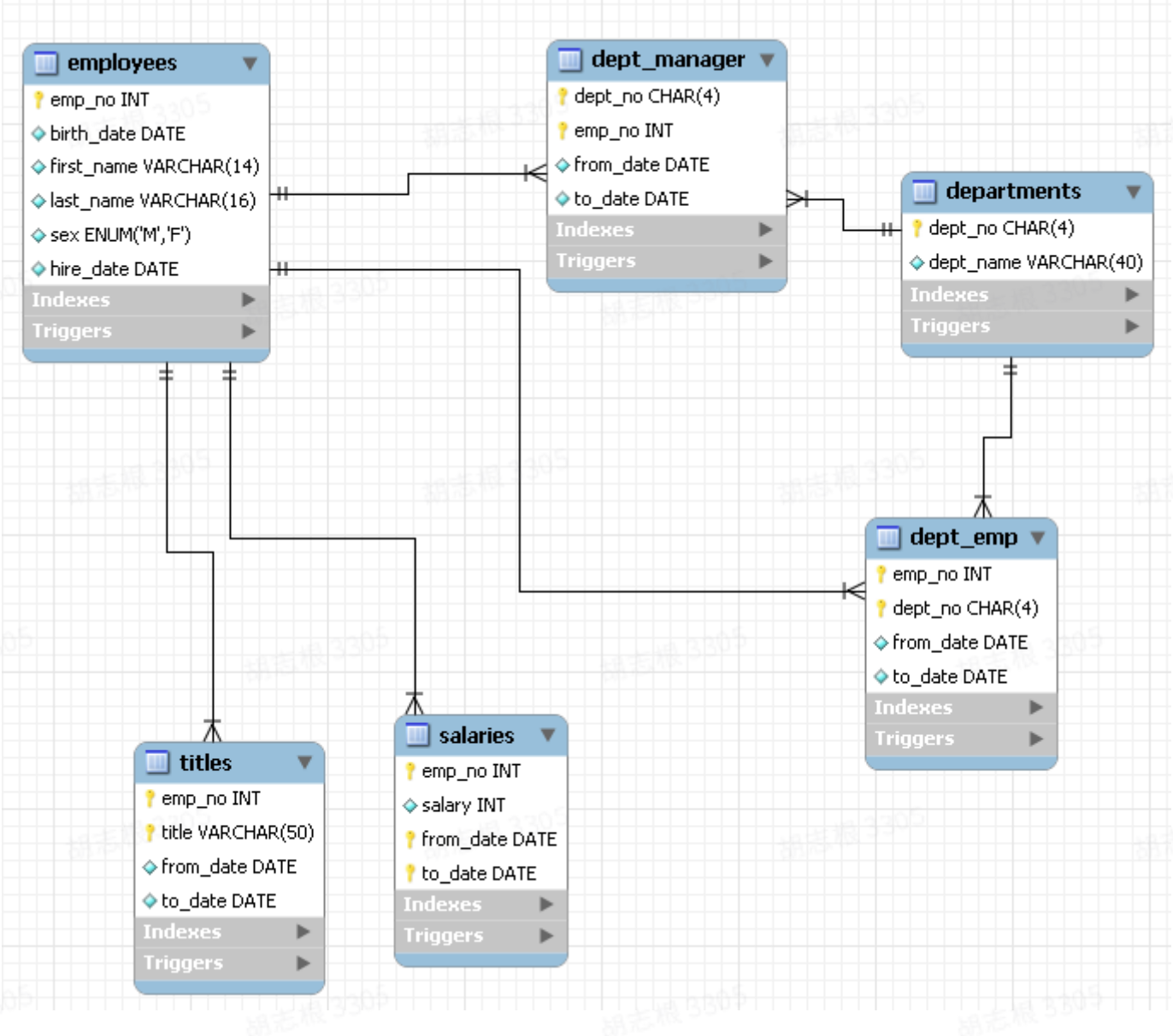
是否明白了为啥会有最左前缀匹配原则了吧？！

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）。本节讨论的高性能索引策略主要属于结构优化范畴。本节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。

示例数据库

为了讨论索引策略，需要一个数据量不算小的数据库作为示例。本文选用MySQL官方文档中提供的示例数据库之一：employees https://github.com/datacharmer/test_db。这个数据库关系复杂度适中，且数据量较大。下图是这个数据库的E-R关系图（引用自MySQL官方手册）：



MySQL官方文档中关于此数据库的页面为https://dev.mysql.com/doc/employee/en/employees-preface.html。里面详细介绍了此数据库，并提供了下载地址和导入方法，如果有兴趣导入此数据库到自己的MySQL可以参考文中内容。

索引使用与相关优化

以employees.titles表为例，下面先查看其上都有哪些索引：

```
1 DESC employees.titles;
2 SHOW INDEX FROM employees.titles;
```

```
MySQL [employees]> desc employees.titles;
+-----------+-------------+------+-----+---------+-------+
| Field     | Type        | Null | Key | Default | Extra |
+-----------+-------------+------+-----+---------+-------+
| emp_no    | int(11)     | NO   | PRI | NULL    |       |
| title     | varchar(50) | NO   | PRI | NULL    |       |
| from_date | date        | NO   | PRI | NULL    |       |
| to_date   | date        | YES  |     | NULL    |       |
+-----------+-------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```

```
MySQL [employees]> SHOW INDEX FROM employees.titles;
+--------+------------+---------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| Table  | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment |
+--------+------------+---------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| titles |          0 | PRIMARY  |            1 | emp_no      | A         |      299390 |     NULL | NULL   |      | BTREE      |         |               |
| titles |          0 | PRIMARY  |            2 | title       | A         |      441654 |     NULL | NULL   |      | BTREE      |         |               |
| titles |          0 | PRIMARY  |            3 | from_date   | A         |      441891 |     NULL | NULL   |      | BTREE      |         |               |
+--------+------------+---------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
3 rows in set (0.01 sec)
```

可以看到具有联合PRIMARY索引（emp_no，title，from_date）

情况一：全列匹配

```sql
EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND title='Senior Engineer' AND from_date='1986-06-26';
```

```
MySQL [employees]> EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND title='Senior Engineer' AND from_date='1986-06-26';
+----+-------------+--------+------------+-------+---------------+---------+---------+-------------------+------+----------+-------+
| id | select_type | table  | partitions | type  | possible_keys | key     | key_len | ref               | rows | filtered | Extra |
+----+-------------+--------+------------+-------+---------------+---------+---------+-------------------+------+----------+-------+
|  1 | SIMPLE      | titles | NULL       | const | PRIMARY       | PRIMARY | 59      | const,const,const |    1 |   100.00 | NULL  |
+----+-------------+--------+------------+-------+---------------+---------+---------+-------------------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

很明显，当按照索引中所有列进行精确匹配（这里精确匹配指“=”或“IN”匹配）时，索引可以被用到。这里有一点需要注意，理论上索引对顺序是敏感的，但是由于MySQL的查询优化器会自动调整where子句的条件顺序以使用适合的索引，例如我们将where中的条件顺序颠倒：

```sql
EXPLAIN SELECT * FROM employees.titles WHERE title='Senior Engineer' AND from_date='1986-06-26' AND emp_no='10001';
```

```
MySQL [employees]> EXPLAIN SELECT * FROM employees.titles WHERE title='Senior Engineer' AND from_date='1986-06-26' AND emp_no='10001';
+----+-------------+--------+------------+-------+---------------+---------+---------+-------------------+------+----------+-------+
| id | select_type | table  | partitions | type  | possible_keys | key     | key_len | ref               | rows | filtered | Extra |
+----+-------------+--------+------------+-------+---------------+---------+---------+-------------------+------+----------+-------+
|  1 | SIMPLE      | titles | NULL       | const | PRIMARY       | PRIMARY | 59      | const,const,const |    1 |   100.00 | NULL  |
+----+-------------+--------+------------+-------+---------------+---------+---------+-------------------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

效果是一样的。

情况二：最左前缀匹配

```sql
EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001';
```

```
MySQL [employees]> EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001';
+----+-------------+--------+------------+------+---------------+---------+---------+-------+------+----------+-------+
| id | select_type | table  | partitions | type | possible_keys | key     | key_len | ref   | rows | filtered | Extra |
+----+-------------+--------+------------+------+---------------+---------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | titles | NULL       | ref  | PRIMARY       | PRIMARY | 4       | const |    1 |   100.00 | NULL  |
+----+-------------+--------+------------+------+---------------+---------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

当查询条件精确匹配索引的左边连续一个或几个列时，如<emp_no>或<emp_no, title>，索引可以被用到，但是只能用到一部分，即条件所组成的最左前缀。上面的查询从分析结果看用到了PRIMARY索引，但是key_len为4，说明只用到了索引的第一列前缀。

```sql
EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND title='Senior Engineer';
```

```
MySQL [employees]> EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND title='Senior Engineer';
+----+-------------+--------+------------+------+---------------+---------+---------+-------------+------+----------+-------+
| id | select_type | table  | partitions | type | possible_keys | key     | key_len | ref         | rows | filtered | Extra |
+----+-------------+--------+------------+------+---------------+---------+---------+-------------+------+----------+-------+
|  1 | SIMPLE      | titles | NULL       | ref  | PRIMARY       | PRIMARY | 56      | const,const |    1 |   100.00 | NULL  |
+----+-------------+--------+------------+------+---------------+---------+---------+-------------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

key_len为56，说明只用到了索引的第一列和第二列前缀。

情况三：查询条件用到了索引中列的精确匹配，但是中间某个条件未提供

```sql
EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND from_date='1986-06-26';
```

```
MySQL [employees]> EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND from_date='1986-06-26';
+----+-------------+--------+------------+------+---------------+---------+---------+-------+------+----------+-------------+
| id | select_type | table  | partitions | type | possible_keys | key     | key_len | ref   | rows | filtered | Extra       |
+----+-------------+--------+------------+------+---------------+---------+---------+-------+------+----------+-------------+
|  1 | SIMPLE      | titles | NULL       | ref  | PRIMARY       | PRIMARY | 4       | const |    1 |    10.00 | Using where |
+----+-------------+--------+------------+------+---------------+---------+---------+-------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

此时索引使用情况和情况二相同，因为title未提供，所以查询只用到了索引的第一列，而后面的from_date虽然也在索引中，但是由于title不存在而无法和左前缀连接，因此需要对结果进行扫描过滤from_date（这里由于emp_no唯一，所以不存在扫描）。

如果想让from_date也使用索引而不是where过滤，可以增加一个辅助索引<emp_no, from_date>，此时上面的查询会使用这个索引。

除此之外，还可以使用一种称之为“隔离列”的优化方法，将emp_no与from_date之间的“坑”填上。

首先我们看下title一共有几种不同的值：

```
SELECT DISTINCT(title) FROM employees.titles;
```

```
+--------------------+
| title              |
+--------------------+
| Senior Engineer    |
| Staff              |
| Engineer           |
| Senior Staff       |
| Assistant Engineer |
| Technique Leader   |
| Manager            |
+--------------------+
7 rows in set (0.21 sec)
```

只有7种。在这种成为“坑”的列值比较少的情况下，可以考虑用“IN”来填补这个“坑”从而形成最左前缀：

```
EXPLAIN SELECT * FROM employees.titles
WHERE emp_no='10001'
AND title IN ('Senior Engineer', 'Staff', 'Engineer', 'Senior Staff', 'Assistant Engineer', 'Technique Leader', 'Manager')
AND from_date='1986-06-26';
```

```
MySQL [employees]> EXPLAIN SELECT * FROM employees.titles
    -> WHERE emp_no='10001'
    -> AND title IN ('Senior Engineer', 'Staff', 'Engineer', 'Senior Staff', 'Assistant Engineer', 'Technique Leader', 'Manager')
    -> AND from_date='1986-06-26';
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
| id | select_type | table  | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra       |
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | titles | NULL       | range | PRIMARY       | PRIMARY | 59      | NULL |    7 |   100.00 | Using where |
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

这次key_len为59，说明索引被用全了，但是从type和rows看出IN实际上执行了一个range查询，这里检查了7个key。看下两种查询的性能比较：

```sql
SET PROFILING=1;

SELECT * FROM employees.titles WHERE emp_no='10001' AND from_date='1986-06-26';

SELECT * FROM employees.titles
WHERE emp_no='10001'
AND title IN ('Senior Engineer', 'Staff', 'Engineer', 'Senior Staff', 'Assistant Engineer', 'Technique Leader', 'Manager')
AND from_date='1986-06-26';

SHOW PROFILES;
```

```
MySQL [employees]> SET PROFILING=1;
Query OK, 0 rows affected, 1 warning (0.00 sec)

MySQL [employees]> SELECT * FROM employees.titles WHERE emp_no='10001' AND from_date='1986-06-26';
+--------+-----------------+------------+------------+
| emp_no | title           | from_date  | to_date    |
+--------+-----------------+------------+------------+
|  10001 | Senior Engineer | 1986-06-26 | 9999-01-01 |
+--------+-----------------+------------+------------+
1 row in set (0.00 sec)

MySQL [employees]> SELECT * FROM employees.titles
    -> WHERE emp_no='10001'
    -> AND title IN ('Senior Engineer', 'Staff', 'Engineer', 'Senior Staff', 'Assistant Engineer', 'Technique Leader', 'Manager')
    -> AND from_date='1986-06-26';
+--------+-----------------+------------+------------+
| emp_no | title           | from_date  | to_date    |
+--------+-----------------+------------+------------+
|  10001 | Senior Engineer | 1986-06-26 | 9999-01-01 |
+--------+-----------------+------------+------------+
1 row in set (0.00 sec)

MySQL [employees]> SHOW PROFILES;
+----------+------------+-----------------------------------------------------------------------------------------------------------------------------------
-------------------------------------+
| Query_ID | Duration   | Query
                                |
+----------+------------+-----------------------------------------------------------------------------------------------------------------------------------
-------------------------------------+
|        1 | 0.00126075 | SELECT * FROM employees.titles WHERE emp_no='10001' AND from_date='1986-06-26'
                                |
|        2 | 0.00067400 | SELECT * FROM employees.titles
WHERE emp_no='10001'
AND title IN ('Senior Engineer', 'Staff', 'Engineer', 'Senior Staff', 'Assistant Engineer', 'Technique Leader', 'Manager')
AND from_date='1986-06-26' |
+----------+------------+-----------------------------------------------------------------------------------------------------------------------------------
-------------------------------------+
2 rows in set, 1 warning (0.00 sec)
```

“填坑”后性能提升了一点。如果经过emp_no筛选后余下很多数据，则后者性能优势会更加明显。当然，如果title的值很多，用填坑就不合适了，必须建立辅助索引。

情况四：查询条件没有指定索引第一列

```sql
EXPLAIN SELECT * FROM employees.titles WHERE from_date='1986-06-26';
```

```
MySQL [employees]> EXPLAIN SELECT * FROM employees.titles WHERE from_date='1986-06-26';
+----+-------------+--------+------------+------+---------------+------+---------+------+--------+----------+-------------+
| id | select_type | table  | partitions | type | possible_keys | key  | key_len | ref  | rows   | filtered | Extra       |
+----+-------------+--------+------------+------+---------------+------+---------+------+--------+----------+-------------+
|  1 | SIMPLE      | titles | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 441891 |    10.00 | Using where |
+----+-------------+--------+------------+------+---------------+------+---------+------+--------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

由于不是最左前缀，索引这样的查询显然用不到索引

情况五：匹配某列的前缀字符串

```sql
EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND title LIKE
'Senior%';

```

```
3 EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND title LIKE
  '%Senior';
4
5 EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND title LIKE
  'Sen%ior';
```

```
MySQL [employees]> EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND title LIKE 'Senior%';
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
| id | select_type | table  | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra       |
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | titles | NULL       | range | PRIMARY       | PRIMARY | 56      | NULL |    1 |   100.00 | Using where |
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.01 sec)

MySQL [employees]> EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND title LIKE '%Senior';
+----+-------------+--------+------------+------+---------------+---------+---------+-------+------+----------+-------------+
| id | select_type | table  | partitions | type | possible_keys | key     | key_len | ref   | rows | filtered | Extra       |
+----+-------------+--------+------------+------+---------------+---------+---------+-------+------+----------+-------------+
|  1 | SIMPLE      | titles | NULL       | ref  | PRIMARY       | PRIMARY | 4       | const |    1 |    11.11 | Using where |
+----+-------------+--------+------------+------+---------------+---------+---------+-------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)

MySQL [employees]> EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND title LIKE 'Sen%ior';
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
| id | select_type | table  | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra       |
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | titles | NULL       | range | PRIMARY       | PRIMARY | 56      | NULL |    1 |   100.00 | Using where |
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

此时可以用到索引，如果通配符%不出现在开头，则可以用到索引，但根据具体情况不同可能只会用其中一个前缀

情况六：范围查询

```
1 EXPLAIN SELECT * FROM employees.titles WHERE emp_no < '10010' and title='Senior
  Engineer';
```

```
MySQL [employees]> EXPLAIN SELECT * FROM employees.titles WHERE emp_no < '10010' and title='Senior Engineer';
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
| id | select_type | table  | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra       |
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | titles | NULL       | range | PRIMARY       | PRIMARY | 4       | NULL |   14 |    10.00 | Using where |
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

范围列可以用到索引（必须是最左前缀），但是范围列后面的列无法用到索引。同时，索引最多用于一个范围列，因此如果查询条件中有两个范围列则无法全用到索引。

```
1 EXPLAIN SELECT * FROM employees.titles
3 WHERE emp_no < '10010'
4 AND title='Senior Engineer'
5 AND from_date BETWEEN '1986-01-01' AND '1986-12-31';
```

```
MySQL [employees]> EXPLAIN SELECT * FROM employees.titles
    -> WHERE emp_no < '10010'
    -> AND title='Senior Engineer'
    -> AND from_date BETWEEN '1986-01-01' AND '1986-12-31';
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
| id | select_type | table  | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra       |
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | titles | NULL       | range | PRIMARY       | PRIMARY | 4       | NULL |   14 |     1.11 | Using where |
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

可以看到索引对第二个范围索引无能为力。这里特别要说明下，那就是MySQL仅用`EXPLAIN`可能无法区分范围索引和多值匹配，因为在type中这两者都显示为range。同时，用了“between”并不意味着就是范围查询，例如下面的查询：

```
1 EXPLAIN SELECT * FROM employees.titles
2 WHERE emp_no BETWEEN '10001' AND '10010'
3 AND title='Senior Engineer'
4 AND from_date BETWEEN '1986-01-01' AND '1986-12-31';
```

```
MySQL [employees]> EXPLAIN SELECT * FROM employees.titles
    -> WHERE emp_no BETWEEN '10001' AND '10010'
    -> AND title='Senior Engineer'
    -> AND from_date BETWEEN '1986-01-01' AND '1986-12-31';
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
| id | select_type | table  | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra       |
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | titles | NULL       | range | PRIMARY       | PRIMARY | 59      | NULL |   15 |     1.11 | Using where |
+----+-------------+--------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

看起来是用了两个范围查询，但作用于emp_no上的“BETWEEN”实际上相当于“IN”，也就是说emp_no实际是多值精确匹配。可以看到这个查询用到了索引全部三个列。因此在MySQL中要谨慎地区分多值匹配和范围匹配，否则会对MySQL的行为产生困惑。

情况七：查询条件中含有函数或表达式。

很不幸，如果查询条件中含有函数或表达式，则MySQL不会为这列使用索引（虽然某些在数学意义上可以使用）。例如：

```
1 EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND left(title,
  6)='Senior';
```

```
MySQL [employees]> EXPLAIN SELECT * FROM employees.titles WHERE emp_no='10001' AND left(title, 6)='Senior';
+----+-------------+--------+------------+------+---------------+---------+---------+-------+------+----------+-------------+
| id | select_type | table  | partitions | type | possible_keys | key     | key_len | ref   | rows | filtered | Extra       |
+----+-------------+--------+------------+------+---------------+---------+---------+-------+------+----------+-------------+
|  1 | SIMPLE      | titles | NULL       | ref  | PRIMARY       | PRIMARY | 4       | const |    1 |   100.00 | Using where |
+----+-------------+--------+------------+------+---------------+---------+---------+-------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

虽然这个查询和情况五中功能相同，但是由于使用了函数left，则无法为title列应用索引，而情况五中用LIKE则可以。

再如：

```
1 EXPLAIN SELECT * FROM employees.titles WHERE emp_no - 1='10000';
```

```
MySQL [employees]> EXPLAIN SELECT * FROM employees.titles WHERE emp_no - 1='10000';
+----+-------------+--------+------------+------+---------------+------+---------+------+--------+----------+-------------+
| id | select_type | table  | partitions | type | possible_keys | key  | key_len | ref  | rows   | filtered | Extra       |
+----+-------------+--------+------------+------+---------------+------+---------+------+--------+----------+-------------+
|  1 | SIMPLE      | titles | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 441891 |   100.00 | Using where |
+----+-------------+--------+------------+------+---------------+------+---------+------+--------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

显然这个查询等价于查询emp_no为10001的函数，但是由于查询条件是一个表达式，MySQL无法为其使用索引。看来MySQL还没有智能到自动优化常量表达式的程度，因此在写查询语句时尽量避免表达式出现在查询中，而是先手工私下代数运算，转换为无表达式的查询语句。

索引选择性与前缀索引

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，MySQL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。

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。（2000记录为界限，未测试）
- 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 Index Selectivity = Cardinality / #T
```

显然选择性的取值范围为(0, 1]，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，上文用到的employees.titles表，如果title字段经常被单独查询，是否需要建索引，我们看一下它的选择性：

```
1 SELECT count(DISTINCT(title))/count(*) AS Selectivity FROM employees.titles;
```

```
MySQL [employees]> SELECT count(DISTINCT(title))/count(*) AS Selectivity FROM employees.titles;
+-------------+
| Selectivity |
+-------------+
|      0.0000 |
+-------------+
1 row in set (0.13 sec)
```

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。

有一种与索引选择性有关的索引优化策略叫做前缀索引，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。

下面以employees.employees表为例介绍前缀索引的选择和使用。

```
MySQL [employees]> show index from employees.employees;
+-----------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| Table     | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment |
+-----------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
| employees |          0 | PRIMARY  |            1 | emp_no      | A         |      299246 |     NULL | NULL   |      | BTREE      |         |               |
+-----------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+------+------------+---------+---------------+
1 row in set (0.00 sec)
```

```
MySQL [employees]> desc employees.employees;
+------------+---------------+------+-----+---------+-------+
| Field      | Type          | Null | Key | Default | Extra |
+------------+---------------+------+-----+---------+-------+
| emp_no     | int(11)       | NO   | PRI | NULL    |       |
| birth_date | date          | NO   |     | NULL    |       |
| first_name | varchar(14)   | NO   |     | NULL    |       |
| last_name  | varchar(16)   | NO   |     | NULL    |       |
| gender     | enum('M','F') | NO   |     | NULL    |       |
| hire_date  | date          | NO   |     | NULL    |       |
+------------+---------------+------+-----+---------+-------+
6 rows in set (0.00 sec)
```

可以看到employees表只有一个索引<emp_no>，那么如果我们想按名字搜索一个人，就只能全表扫描了：

```
1 EXPLAIN SELECT * FROM employees.employees WHERE first_name='Eric' AND
  last_name='Anido';
```

```
MySQL [employees]> EXPLAIN SELECT * FROM employees.employees WHERE first_name='Eric' AND last_name='Anido';
+----+-------------+-----------+------------+------+---------------+------+---------+------+--------+----------+-------------+
| id | select_type | table     | partitions | type | possible_keys | key  | key_len | ref  | rows   | filtered | Extra       |
+----+-------------+-----------+------------+------+---------------+------+---------+------+--------+----------+-------------+
|  1 | SIMPLE      | employees | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 299246 |     1.00 | Using where |
+----+-------------+-----------+------------+------+---------------+------+---------+------+--------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

如果频繁按名字搜索员工，这样显然效率很低，因此我们可以考虑建索引。有两种选择，建<first_name>或<first_name, last_name>，看下两个索引的选择性：

```
SELECT count(DISTINCT(first_name))/count(*) AS Selectivity FROM
employees.employees;
SELECT count(DISTINCT(concat(first_name, last_name)))/count(*) AS Selectivity FROM
employees.employees;
```

```
MySQL [employees]> SELECT count(DISTINCT(first_name))/count(*) AS Selectivity FROM employees.employees;
+-------------+
| Selectivity |
+-------------+
|      0.0042 |
+-------------+
1 row in set (0.15 sec)

MySQL [employees]> SELECT count(DISTINCT(concat(first_name, last_name)))/count(*) AS Selectivity FROM employees.employees;
+-------------+
| Selectivity |
+-------------+
|      0.9313 |
+-------------+
1 row in set (0.42 sec)
```

<first_name>显然选择性太低，<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：

```
MySQL [employees]> SELECT count(DISTINCT(concat(first_name, left(last_name, 3))))/count(*) AS Selectivity FROM employees.employees;
+-------------+
| Selectivity |
+-------------+
|      0.7879 |
+-------------+
1 row in set (0.37 sec)
```

选择性还不错，但离0.9313还是有点距离，那么把last_name前缀加到4：

```
MySQL [employees]> SELECT count(DISTINCT(concat(first_name, left(last_name, 4))))/count(*) AS Selectivity FROM employees.employees;
+-------------+
| Selectivity |
+-------------+
|      0.9007 |
+-------------+
1 row in set (0.38 sec)
```

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：

```
ALTER TABLE employees.employees
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：

```
MySQL [employees]> use employees;
Database changed
MySQL [employees]> SET PROFILING=1;
Query OK, 0 rows affected, 1 warning (0.00 sec)

MySQL [employees]> SELECT * FROM employees.employees WHERE first_name='Eric' AND last_name='Anido';
+--------+------------+------------+-----------+--------+------------+
| emp_no | birth_date | first_name | last_name | gender | hire_date  |
+--------+------------+------------+-----------+--------+------------+
|  18454 | 1955-02-28 | Eric       | Anido     | M      | 1988-07-18 |
+--------+------------+------------+-----------+--------+------------+
1 row in set (0.09 sec)

MySQL [employees]> ALTER TABLE employees.employees
    -> ADD INDEX `first_name_last_name4` (first_name, last_name(4));
Query OK, 0 rows affected (4.53 sec)
Records: 0  Duplicates: 0  Warnings: 0

MySQL [employees]> SELECT * FROM employees.employees WHERE first_name='Eric' AND last_name='Anido';
+--------+------------+------------+-----------+--------+------------+
| emp_no | birth_date | first_name | last_name | gender | hire_date  |
+--------+------------+------------+-----------+--------+------------+
|  18454 | 1955-02-28 | Eric       | Anido     | M      | 1988-07-18 |
+--------+------------+------------+-----------+--------+------------+
1 row in set (0.01 sec)
```

```
MySQL [employees]> SHOW PROFILES;
+----------+------------+-----------------------------------------------------------------------------------------------------+
| Query_ID | Duration   | Query                                                                                               |
+----------+------------+-----------------------------------------------------------------------------------------------------+
|        1 | 0.08834025 | SELECT * FROM employees.employees WHERE first_name='Eric' AND last_name='Anido'                     |
|        2 | 4.53062525 | ALTER TABLE employees.employees
ADD INDEX `first_name_last_name4` (first_name, last_name(4)) |
|        3 | 0.00107925 | SELECT * FROM employees.employees WHERE first_name='Eric' AND last_name='Anido'                     |
|        4 | 0.00057800 | SELECT * FROM employees.employees WHERE first_name='Eric' AND last_name='Anido'                     |
|        5 | 0.00086725 | SELECT * FROM employees.employees WHERE first_name='Eric' AND last_name='Anido'                     |
+----------+------------+-----------------------------------------------------------------------------------------------------+
5 rows in set, 1 warning (0.00 sec)
```

性能的提升是显著的，查询速度提高了120多倍。
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）。

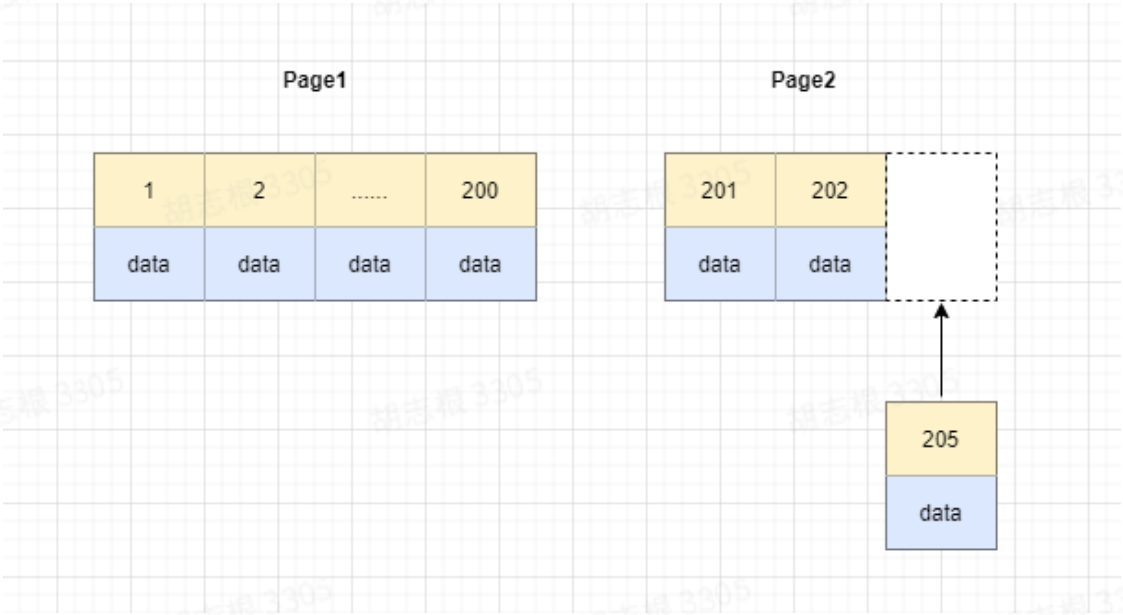
InnoDB的主键选择与插入优化

在使用InnoDB存储引擎时，如果没有特别的需要，请永远使用一个与业务无关的自增字段作为主键。经常看到有帖子或博客讨论主键选择问题，有人建议使用业务无关的自增主键，有人觉得没有必要，完全可以使用如学号或身份证号这种唯一字段作为主键。不论支持哪种论点，大多数论据都是业务层面的。如果从数据库索引优化角度看，使用InnoDB引擎而不使用自增主键绝对是一个糟糕的主意。

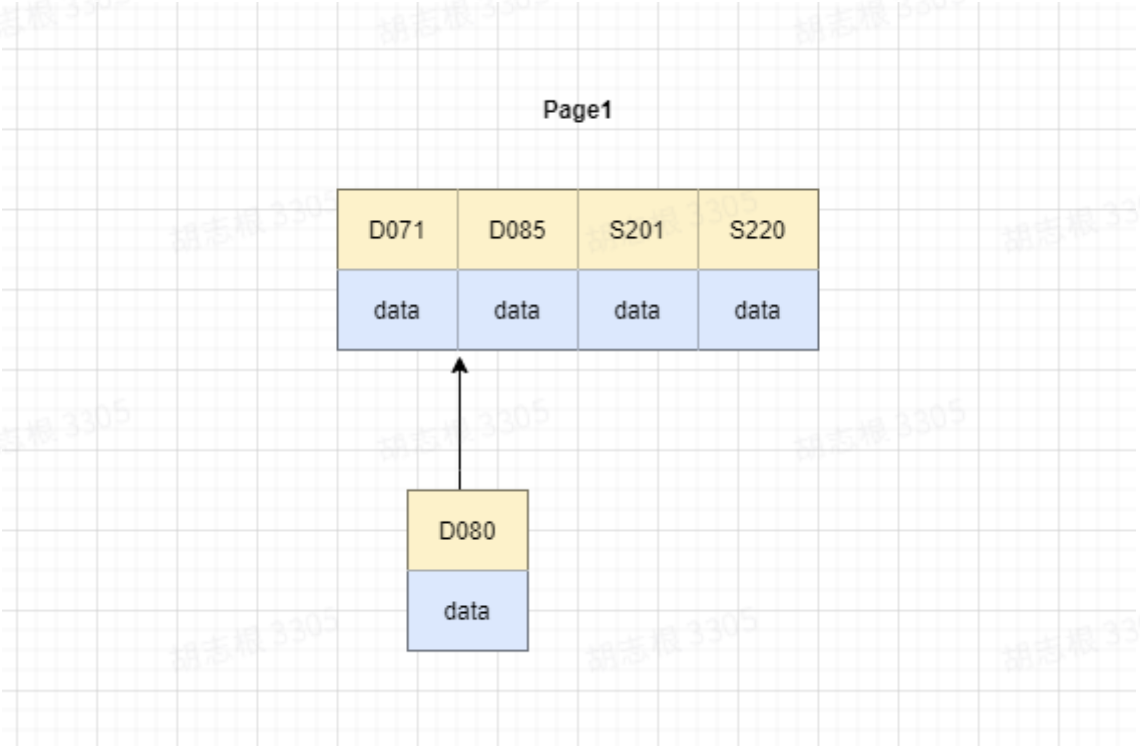
上文讨论过InnoDB的索引实现，InnoDB使用聚集索引，数据记录本身被存于主索引（一颗B+Tree）的叶子节点上。这就要求同一个叶子节点内（大小为一个内存页或磁盘页）的各条数据记录按主键顺

序存放，因此每当有一条新的记录插入时，MySQL会根据其主键将其插入适当的节点和位置，如果页面达到装载因子（InnoDB默认为15/16），则开辟一个新的页（节点）。

如果表使用自增主键，那么每次插入新的记录，记录就会顺序添加到当前索引节点的后续位置，当一页写满，就会自动开辟一个新的页。如下图所示：



这样就会形成一个紧凑的索引结构，近似顺序填满。由于每次插入时也不需要移动已有数据，因此效率很高，也不会增加很多开销在维护索引上。
如果使用非自增主键（如果身份证号或学号等），由于每次插入主键的值近似于随机，因此每次新纪录都要被插到现有索引页得中间某个位置：



此时MySQL不得不为了将新记录插到合适位置而移动数据，甚至目标页面可能已经被回写到磁盘上而从缓存中清掉，此时又要从磁盘上读回来，这增加了很多开销，同时频繁的移动、分页操作造成了大量的碎片，得到了不够紧凑的索引结构，后续不得不通过OPTIMIZE TABLE来重建表并优化填充页面。
因此，只要可以，请尽量在InnoDB上采用自增字段做主键。

未完待续

其实数据库索引调优是一项技术活，不能仅仅靠理论，因为实际情况千变万化，而且MySQL本身存在很复杂的机制，如查询优化策略和各种引擎的实现差异等都会使情况变得更加复杂。但同时这些理论是索引调优的基础，只有在明白理论的基础上，才能对调优策略进行合理推断并了解其背后的机制，然后结合实践中不断的实验和摸索，从而真正达到高效使用MySQL索引的目的。

另外，MySQL索引及其优化涵盖范围非常广，本文只是涉及到其中一部分。如与排序（ORDER BY）相关的索引优化及覆盖索引（Covering index）的话题本文并未涉及，同时除B-Tree索引外MySQL还根据不同引擎支持的哈希索引、全文索引等等本文也并未涉及。有兴趣的同学可以自行查阅资料学习。

参考文献

[1] 高性能MySQL：https://book.douban.com/subject/23008813/
[2] MySQL技术内幕-InnoDB存储引擎：https://book.douban.com/subject/24708143/