**数据库索引的通俗理解**

**为什么需要索引？**

**「索引就像书的目录， 通过书的目录就准确的定位到了书籍具体的内容」，**

数据在磁盘上是**以块的形式存储的**。为确保对磁盘操作的原子性，访问数据的时候会一并访问所有数据块。磁盘上的这些数据块与链表类似，即它们都**包含一个数据段和一个指针，指针指向下一个节点（数据块）的内存地址，**而且它们都不需要连续存储（即逻辑上相邻的数据块在物理上可以相隔很远）。

举个例子来讲，我们有一个数据表User.为了简便，这个表没有主键。

|  |  |  |  |
| --- | --- | --- | --- |
| Identity | Name | Age | Grade |
| 1 | Robin | 28 | 90 |
| 5 | Lilei | 26 | 60 |
| 3 | Hanmei | 25 | 50 |
| 4 | Lucy | 27 | 66 |
| 2 | Lily | 29 | 80 |

虽然这些数据都存在于一个User表中，但是物理上，这些数据可能存储在分散的数据块中。

查找Lily这个人的信息, 已知Lily的Identity为2，

select \* fromUser where Identity= 2.

在查找的时候，首先找到这个表的第一条记录所在的数据库地址，然后发现Identity为1，并不是所需要的值，然后在这个数据库的底端，找到了下一个数据块的地址。（这个类似于链表），如此一来，**查询了5次才找到了所需要的值**。（为了简单起见，我们考虑Identity不能有重复值）

**为了加快搜索速度，这里就出现了索引。索引是对某个字段进行排序的一种方式。对表中的某个字段建立索引会创建另一种数据结构(树)，其中保存着字段的值，每个值又指向与它相关的记录。这种索引的数据结构是经过排序的，因而可以对其执行二分查找。**

对上个表的Identity字段进行索引，就是在数据库存储空间上创建一块专用的控件，把User表的所有的Identity字段的值拿出来放到这里，并且对这些值进行排序，并且每个值都携带着这个Identity对应的行所在数据块的地址。因为Identity是进过排序的，按照一定的数据结构存储的，所以数据库引擎在查找的时候，

比如说查找identity为5，引擎就会计算，5大概在整个排序结构的大致地方，然后到那里去拿出这个值看看是不是，不是的话就再次相应的向左或者向右移动去寻找。（这里用到的知识都是大学时候的数据结构的知识，**二分法查找**，相对于毫无头绪的一个一个的查找，二分法的查找速度明显的提高，达到了log2 N，其实这有多快我也不明白，反正就记得当时学的时候，确实是比一般查找快多了。）

通俗的来讲，就是根据你指定的列，建立一个遵循一定数据结构的区域，这些区域可以**快速定位到相应数据库字段所在的磁盘地址**。

索引的好处是特别明显的，那就是大大的提高了查询的速度。但是相对应的也带来了一些不好的地方。

第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

最后还有一点需要注意的是，我们在数据库上对于某个字段建立了索引，那么什么情况下才走索引呢？

比如 select \* from User where Identity= 2 这条语句，是走索引查询的。因为是否走索引取决于这条查询语句的where子句。

**数据库引擎发现你的where语句中有identity，那么就会从identity的索引数据结构中进行检索**。

曾经看到有人说select \*会降低检索速度，这个跟索引没关系，select \* 降低检索速度，是因为从数据库服务器端到客户端的网络传输是有时间的**，select \* 中难免包含着不必要的字段，所以传输起来会比较慢**。

上述讲到了存储索引用的是某种数据结构类型，接下来将说一下具体索引的存储用到了什么数据结构的类型：

索引原理必须清楚一种数据结构**「平衡树」(非二叉)**，也就是b tree或者 b+ tree，重要的事情说三遍：“平衡树，平衡树，平衡树”。当然， 有的数据库也使用哈希桶作用索引的数据结构 ， 然而， 主流的RDBMS都是把**平衡树**当做数据表默认的索引数据结构的。

# 关于索引的B tree B-tree B+tree B\*tree 详解结构图

索引分为聚簇索引和非聚簇索引。

以一本英文课本为例，要找第8课，直接翻书，若先翻到第5课，则往后翻，再翻到第10课，则又往前翻。这本书本身就是一个索引，**即“聚簇索引”**。

如果要找"fire”这个单词，会翻到书后面的附录，这个附录是按字母排序的，找到F字母那一块，再找到"fire”，对应的会是它在第几课。这个附录，**为“非聚簇索引”**。

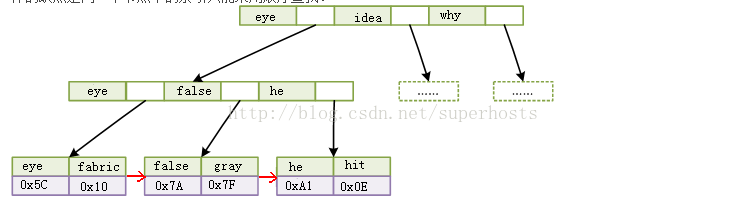
由此可见，聚簇索引，索引的顺序就是数据存放的顺序，所以，很容易理解，**一张数据表只能有一个聚簇索引**。

**聚簇索引要比非聚簇索引查询效率高很多，特别是范围查询的时候**。所以，至于聚簇索引到底应该为主键，还是其他字段，这个可以再讨论。

**1、MYSQL的索引**

mysql中，拥有不同的存储引擎，不同的存储引擎对索引的实现方式不同，大致说下MyISAM和InnoDB两种存储引擎。

MyISAM的B+Tree的叶子节点上的data，并不是数据本身，**而是数据存放的地址**。主索引和辅助索引没啥区别，只是主索引中的key一定得是唯一的。这里的索引都是**非聚簇索引**。MyISAM还采用压缩机制存储索引，比如，第一个索引为“her”，第二个索引为“here”，那么第二个索引会被存储为“3,e”，这样的缺点是**同一个节点中的索引只能采用顺序查找。**



InnoDB的数据文件本身就是索引文件，B+Tree的叶子节点上的**data就是数据本身，key为主键，这是聚簇索引。**

聚簇索引，叶子节点上的data是主键(所以聚簇索引的key，不能过长)。为什么存放的主键，而不是记录所在地址呢，理由相当简单，因为记录所在地址并不能保证**一定不会变**，但主键可以保证。

至于为什么主键通常建议使用自增id呢？

**2、聚簇索引**

**聚簇索引的数据的物理存放顺序与索引顺序是一致的**，即：**只要索引是相邻的，那么对应的数据一定也是相邻地存放在磁盘上的**。如果主键不是自增id，那么可以想象，它会干些什么，不断地调整数据的物理地址、分页，当然也有其他一些措施来减少这些操作，但却无法彻底避免。但，如果是自增的，那就简单了，它只需要一页一页地写，索引结构相对紧凑，磁盘碎片少，效率也高。

聚簇索引不但在检索上可以大大滴提高效率，在数据读取上也一样。比如：需要查询f~t的所有单词。

**一个使用MyISAM的主索引(非聚簇索引)，一个使用InnoDB的聚簇索引。**

两种索引的B+Tree检索时间一样，但读取时却有了差异。

因为MyISAM的主索引并非聚簇索引，那么他的数据的物理地址必然是凌乱的，拿到这些物理地址，**按照合适的算法进行I/O读取，于是开始不停的寻道不停的旋转。聚簇索引则只需一次I/O。**

不过，如果涉及到大数据量的排序、全表扫描、count之类的操作的话，还是MyISAM占优势些，**因为索引所占空间小**，这些操作是需要在内存中完成的。

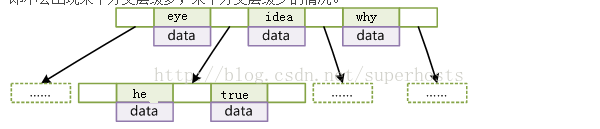
鉴于聚簇索引的范围查询效率，很多人认为使用主键作为聚簇索引太多浪费，毕竟几乎**不会使用主键进行范围查询**。但若再考虑到聚簇索引的存储，就不好定论了。

## 学习笔记】mysql索引原理之B+/-Tree

索引，是为了更快的查询数据，查询算法有很多，对应的数据结构也不少，数据库常用的索引数据结构一般为B+Tree。

**1、B-Tree**

关于B-Tree的官方定义个人觉得比较难懂，通俗一点就是举个例子。假如：一本英文字典，单词+详细解释组成了一条记录，现在需要索引单词，那么以单词为key，单词+详细解释为data，B-Tree就是以一个二元组{key,data}来定义一条记录。如果一个节点有3条记录，那么会有对应的4个指针，用以指向下一个节点。B-Tree是**有序且平衡的，所有叶子节点在同一层，**即不会出现某个分支层级多，某个分支层级少的情况。

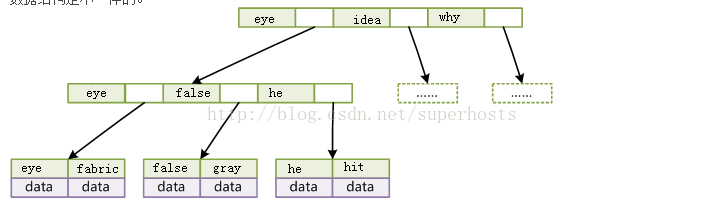


因为B-Tree是有序的，所以它的查找就简单了，先从根节点开始二分查找，**找到则返回节点；否则沿着区间指针查找下一个节点**。

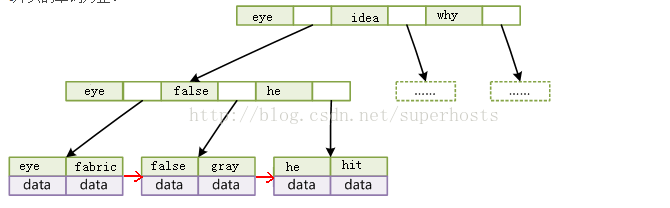
**2、B+Tree**

与B-Tree不同的是，B+Tree每个节点只有key，没有data；而且叶子节点没有指针。

也就是说B+Tree的叶子节点和内节点的数据结构是不一样的。



一般数据库采用的是B+Tree，而且经过了一些优化，比如在**叶子节点上增加了顺序访问指针**，提高区间查询效率。比如：查询首字母为f~t的所有单词。那么只需查到**f开头的第一个单词fabric**，然后沿着叶子节点的开始遍历，直到找到最后一个以t开头的单词为止。



简单介绍了B-/+Tree，至于众多数据结构中，**为何数据库索引不选择BTree，而且选择B+Tree，下面从计算机存储原理方面简单说说。**

### 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。

内节点不存储data，只存储key；叶子节点不存储指针。

图3是一个简单的B+Tree示意。



图3

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，但是每个节点的域和上限是一致的，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。

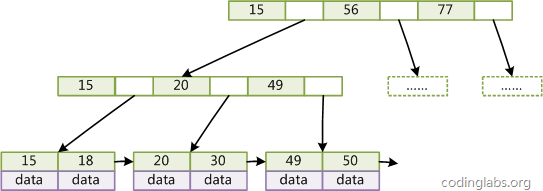


图4

如图4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4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。

**3、读内存和读磁盘**

内存读取和磁盘读取的效率是相差很大的。

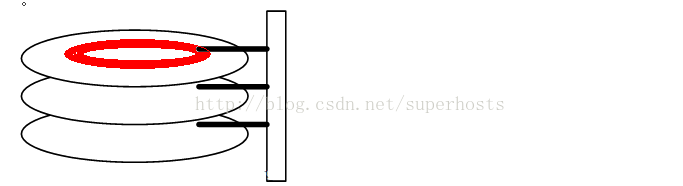
简单点说说内存读取，内存是由一系列的存储单元组成的，每个存储单元存储固定大小的数据，且有一个唯一地址。

**当需要读内存时，将地址信号放到地址总线上传给内存，内存解析信号并定位到存储单元，然后把该存储单元上的数据放到数据总线上，回传**。

写内存时，系统将要写入的数据和单元地址分别放到数据总线和地址总线上，内存读取两个总线的内容，做相应的写操作。

**内存存取效率，跟次数有关**，读取先后无关。先读取A数据还是后读取A数据不会影响存取效率。而磁盘存取就不一样了，磁盘I/O涉及机械操作。

磁盘是由大小相同且同轴的圆形盘片组成，磁盘可以转动(各个磁盘须同时转动)。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。磁头不动，磁盘转动，但磁臂可以前后动，用于读取不同磁道上的数据。磁道就是以盘片为中心划分出来的一系列同心环(如图标红那圈)。**磁道又划分为一个个小段，叫扇区，是磁盘的最小存储单元**。

  
 磁盘读取时，系统**将数据逻辑地址传给磁盘**，磁盘的控制电路会解析出物理地址，即哪个磁道哪个扇区。于是**磁头需要前后移动到对应的磁道**，消耗的时间叫寻道时间，然后**磁盘旋转将对应的扇区转到磁头下**，消耗的时间叫旋转时间。所以，适当的操作顺序和数据存放可以**减少寻道时间和旋转时间**。

为了尽量减少I/O操作，磁盘读取每次都会预读，大小通常为页的整数倍。即使只需要读取一个字节，磁盘也会读取一页的数据(通常为4K)放入内存，内存与磁盘以页为单位交换数据。因为局部性原理认为，通常一个数据被用到，其附近的数据也会立马被用到。

**4、检索性能分析**

B-Tree：如果一次检索需要访问4个节点，数据库系统设计者利用磁盘预读原理，把节点的大小设计为一个页，那读取一个节点只需要一次I/O操作，完成这次检索操作，最多需要3次I/O(根节点常驻内存)。**数据记录越小，每个节点存放的数据就越多，树的高度也就越小，I/O操作就少了，检索效率也就上去了。**

B+Tree：内节点**只存key，大大滴减少了内节点的大小，那么每个节点就可以存放更多的记录，树的更矮了，I/O操作更少了。所以B+Tree拥有更好的性能。**

**5、其他索引方式**

散列索引：通过HASH来定位的一种索引，这种索引用的较少，通过用于单值查询。**InnoDB的自适应索引就是HASH索引**。

**位图索引：字段值固定且少，**比如性别、状态。在同时对多个这样的字段and/or查询时，效率极高，**直接按位与/或就可以得到结果了。所以，应用范围局限(bit)**

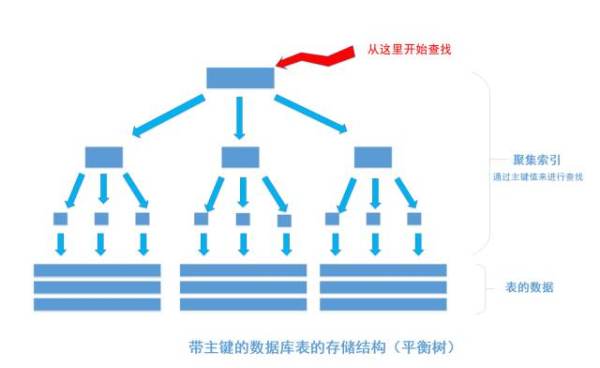
**再深入：**

我们平时建表的时候都会为表加上主键， 在某些关系数据库中， 如果建表时不指定主键，数据库会拒绝建表的语句执行。

事实上， 一个加了主键的表，并不能被称之为「表」。**一个没加主键的表，它的数据无序的放置在磁盘存储器上，一行一行的排列的很整齐**， 跟我认知中的「表」很接近。

如果给表**上了主键，那么表在磁盘上的存储结构就由整齐排列的结构转变成了树状结构，**也就是上面说的「平衡树」结构，换句话说，就是整个表就变成了一个索引。没错， 再说一遍， **整个表变成了一个索引，也就是所谓的「聚集索引」**。

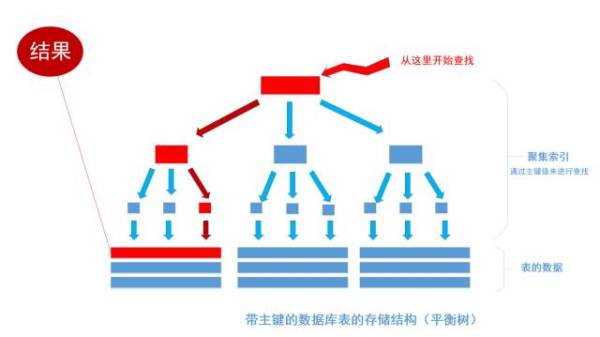
这就是为什么一个表只能有一个主键， 一个表只能有一个「聚集索引」，因为**主键的作用就是把「表」的数据格式转换成「索引（平衡树）」的格式放置**。



上图就是带有主键的表（聚集索引）的结构图。图画的不是很好， 将就着看。其中树的所有结点（底部除外）的数据都是由主键字段中的数据构成，**也就是通常我们指定主键的id字段**。最下面部分是真正表中的数据。 假如我们执行一个SQL语句：

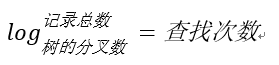
select \* from table where id = 1256;

首先根据索引定位到1256这个值所在的叶结点，然后再通过叶结点取到id等于1256的数据行。 这里不讲解平衡树的运行细节， 但是从上图能看出，树一共有三层， 从根节点至叶节点只需要经过三次查找就能得到结果。如下图



假如一张表有一亿条数据 ，需要查找其中某一条数据，按照常规逻辑， 一条一条的去匹配的话， 最坏的情况下需要匹配一亿次才能得到结果，用大O标记法就是O(n)最坏时间复杂度，这是无法接受的，而且这一亿条数据显然不能一次性读入内存供程序使用， 因此， 这一亿次匹配在不经缓存优化的情况下就是一亿次IO开销，以现在磁盘的IO能力和CPU的运算能力， 有可能需要几个月才能得出结果 。如果把这张表转换成平衡树结构（一棵非常茂盛和节点非常多的树），**假设这棵树有10层，那么只需要10次IO开销就能查找到所需要的数据， 速度以指数级别提升，用大O标记法就是O(log n)，n是记录总树，底数是树的分叉数，结果就是树的层次数。**换言之，查找次数是以树的分叉数为底，记录总数的对数，用公式来表示就是

假如一张表有一亿条数据 ，需要查找其中某一条数据，按照常规逻辑， 一条一条的去匹配的话， 最坏的情况下需要匹配一亿次才能得到结果，用大O标记法就是O(n)最坏时间复杂度，这是无法接受的，而且这一亿条数据显然不能一次性读入内存供程序使用， 因此， 这一亿次匹配在不经缓存优化的情况下就是一亿次IO开销，以现在磁盘的IO能力和CPU的运算能力， 有可能需要几个月才能得出结果 。如果把这张表转换成平衡树结构（一棵非常茂盛和节点非常多的树），假设这棵树有10层，那么只需要10次IO开销就能查找到所需要的数据， 速度以指数级别提升，用大O标记法就是O(log n)，n是记录总树，底数是树的分叉数，结果就是树的层次数。换言之，查找次数是以树的分叉数为底，记录总数的对数，用公式来表示就是

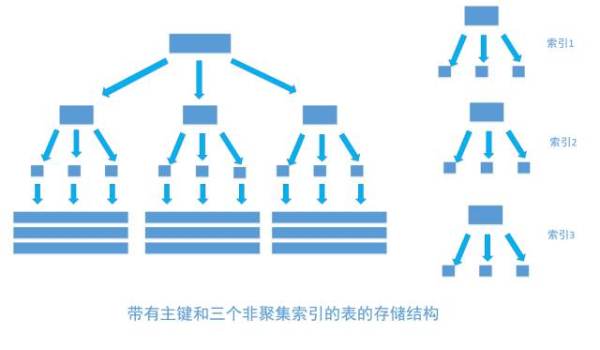


用程序来表示就是Math.Log(100000000,10)，100000000是记录数，10是树的分叉数（真实环境下分叉数远不止10）， 结果就是查找次数，这里的结果从亿降到了个位数。**因此，利用索引会使数据库查询有惊人的性能提升。**

然而， 事物都是有两面的， 索引能让数据库查询数据的速度上升， 而使写入数据的速度下降，原因很简单的， 因为平衡树这个结构必须一直维持在一个正确的状态， **增删改数据都会改变平衡树各节点中的索引数据内容，破坏树结构**， 因此，**在每次数据改变时， DBMS必须去重新梳理树（索引）的结构以确保它的正确**，这会带来不小的性能开销，也就是为什么索引会给查询以外的操作带来副作用的原因。

讲完聚集索引 ， 接下来聊一下**非聚集索引**， 也就是我们平时经常提起和使用的常规索引。

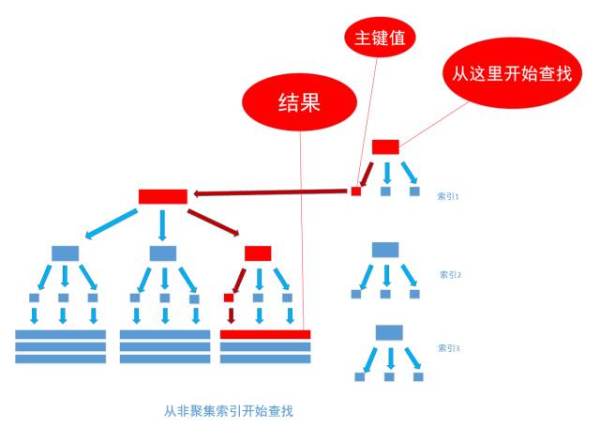
非聚集索引和聚集索引一样， 同样是采用平衡树作为索引的数据结构。索引树结构中各节点的值来自于表中的索引字段， 假如给user表的name字段加上索引 ， 那么索引就是由name字段中的值构成，在数据改变时， DBMS需要一直维护索引结构的正确性。如果给表中多个字段加上索引 ， 那么就**会出现多个独立的索引结构**，**每个索引（非聚集索引）互相之间不存在关联**。 如下图



**每次给字段建一个新索引， 字段中的数据就会被复制一份出来**， 用于生成索引。

因此， 给表添加索引，会增加表的体积， 占用磁盘存储空间。

非聚集索引和聚集索引的区别在于， **通过聚集索引可以查到需要查找的数据， 而通过非聚集索引可以查到记录对应的主键值（其实就是叶子节点data上面的地址指向的那块区域）， 再使用主键的值通过聚集索引查找到需要的数据，**如下图



不管以任何方式查询表， **最终都会利用主键通过聚集索引来定位到数据，** 聚集索引**（主键）是通往真实数据所在的唯一路径**。

然而， 有一种例外可以不使用聚集索引就能查询出所需要的数据， 这种非主流的方法 称之为**「覆盖索引」**查询， 也就是平时所说的复合索引或者多字段索引查询。 文章上面的内容已经指出， 当为字段建立索引以后， **字段中的内容会被同步到索引之中， 如果为一个索引指定两个字段， 那么这个两个字段的内容都会被同步至索引之中。**

先看下面这个SQL语句

//建立索引

create index index\_birthday on user\_info(birthday);

//查询生日在1991年11月1日出生用户的用户名

select user\_name from user\_info where birthday = '1991-11-1'

这句SQL语句的执行过程如下

首先，通过非聚集索引index\_birthday查找birthday等于1991-11-1的所有记录的主键ID值

然后，通过得到的主键ID值执行聚集索引查找，找到主键ID值对就的真实数据（数据行）存储的位置

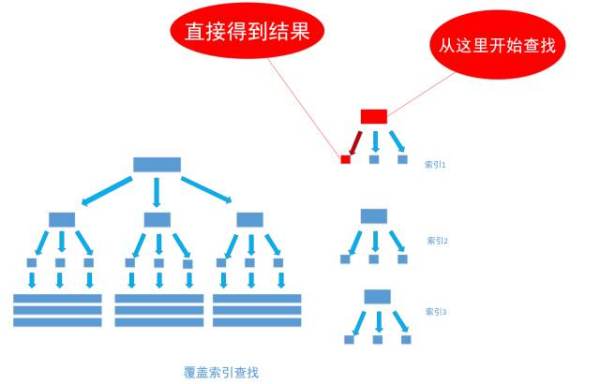
最后， 从得到的真实数据中取得user\_name字段的值返回， 也就是取得最终的结果

我们把birthday字段上的索引**改成双字段的覆盖索引**

create index index\_birthday\_and\_user\_name on user\_info(birthday, user\_name);

这句SQL语句的执行过程就会变为

通过非聚集索引**index\_birthday\_and\_user\_name**查找birthday等于1991-11-1的叶节点的内容，然而， 叶节点中除了有user\_name表主键ID的值以外， user\_name字段的值也在里面， 因此不需要通过主键ID值的查找数据行的真实所在， 直接取得叶节点**中user\_name的值返回即可**。 通过这种覆盖索引直接查找的方式， 可以省略不使用覆盖索引查找的后面两个步骤， **大大的提高了查询性能，**如下图



数据库索引的大致工作原理就是像文中所述， 然而细节方面可能会略有偏差，这但并不会对概念阐述的结果产生影响 。

**一个表只有一个聚集索引（主键），可以有多个非聚集索引。**