索引的实现原理 - timer_gao的博客 - CSDN博客

blog.csdn.net 已更新2019年4月5日

索引的实现原理

转

2017年09月17日 21:47:38 timer_gao 阅读数：24742

这篇文章是介绍MySQL数据库中的索引是如何根据需求一步步演变最终成为B+树结构的以及针对B+树索引的查询，插入，删除，更新等操作的处理方法。Oracle和DB2数据库索引的实现基本上也是大同小异的。文章写得很通俗易懂，就转在这了。关于B+树和索引内部结构可以参考：《B 树、B- 树、B+ 树和B* 树》和《深入理解DB2索引（Index）》。

00 – 背景知识

- B-Tree & B+Tree

http://en.wikipedia.org/wiki/B%2B_tree
http://en.wikipedia.org/wiki/B-tree

- 折半查找(Binary Search)

http://en.wikipedia.org/wiki/Binary_search_algorithm

- 数据库的性能问题

A. 磁盘IO性能非常低，严重的影响数据库系统的性能。
B. 磁盘顺序读写比随机读写的性能高很多。

– 数据的基本存储结构

A. 磁盘空间被划分为许多大小相同的块（Block）或者页(Page).
B. 一个表的这些数据块以链表的方式串联在一起。
C. 数据是以行（Row）为单位一行一行的存放在磁盘上的块中,如图所示.
D. 在访问数据时，一次从磁盘中读出或者写入至少一个完整的Block。

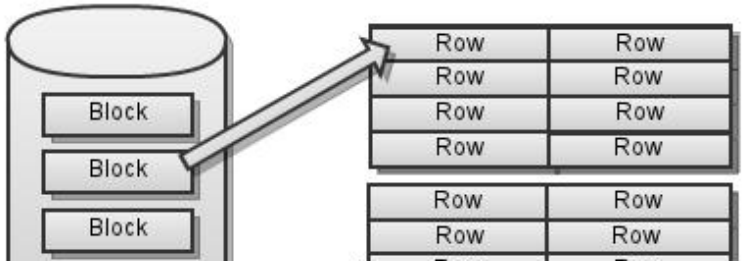


Fig. 1

01 – 数据基本操作的实现

基本操作包括：INSERT、UPDATE、DELETE、SELECT。

- SELECT

A. 定位数据
B. 读出数据所在的块，对数据加工
C. 返回数据给用户

- UPDATE、DELETE

A. 定位数据
B. 读出数据所在的块，修改数据
C. 写回磁盘

- INSERT

A. 定位数据要插入的页（如果数据需要排序）
B. 读出要插入的数据页，插入数据.
C. 写回磁盘

如何定位数据?

- 表扫描(Table Scan)

A. 从磁盘中依次读出所有的数据块，一行一行的进行数据匹配。
B. 时间复杂度 是O(n)， 如果所有的数据占用了100个块。尽管只查询一行数据，
也需要读出所有100个块的数据。
C. 需要大量的磁盘IO操作，极大的影响了数据定位的性能。

因为数据定位操作是所有数据操作必须的操作，数据定位操作的效率会直接影响所有的数据操作的效率。

因此我们开始思考，如何来减少磁盘的IO?

- 减少磁盘IO

A. 减少数据占用的磁盘空间
压缩算法、优化数据存储结构
B. 减少访问数据的总量
读出或写入的数据中，有一部分是数据操作所必须的，这部分称作有效数据。剩余的部分则不是数据操作必须的数据，称为无效数据。例如，查询姓名是‘张三’的记录。那么这条记录是有效记录，其他记录则是无效记录。我们要努力减少无效数据的访问。

02 – 索引的产生

- 键(Key)

首先，我们发现在多数情况下，定位操作并不需要匹配整行数据。而是很规律的只匹配某一个
或几个列的值。 例如，图中第1列就可以用来确定一条记录。这些用来确定一条数据的列，

统
称为**键(Key)**.

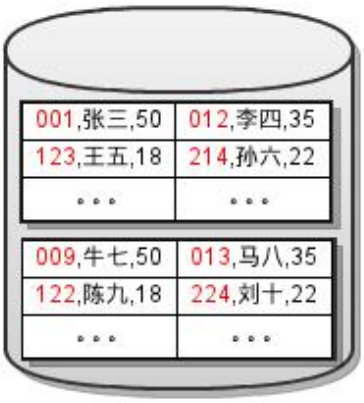


Fig. 2

- Dense Index

根据减少无效数据访问的原则，我们将键的值拿过来存放到独立的块中。并且为每一个键值添
加一个指针， 指向原来的数据块。如图所示，

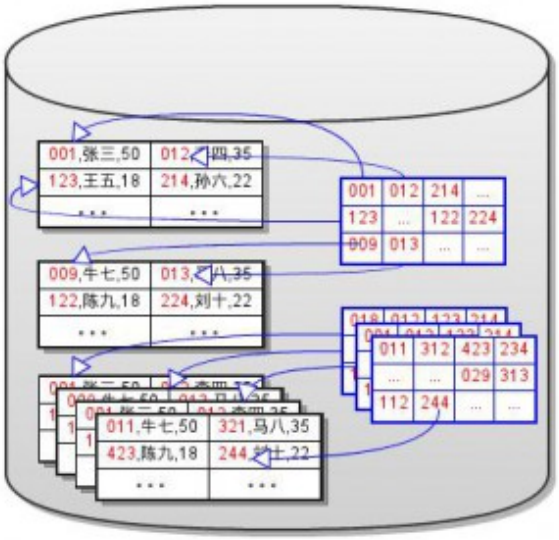


Fig. 3

这就是‘索引’的祖先**Dense Index**. 当进行定位操作时，不再进行表扫描。而是进行**索引扫描(Index Scan)**，依次读出所有的索引块，进行键值的匹配。当找到匹配的键值后，根据该行的指针直接读取对应的数据块，进行操作。假设一个块中能存储100行数据，10,000,000行的数据需要100,000个块的存储空间。假设键值列（+指针）占用一行数据1/10的空间。那么大约需要10,000个块来存储Dense索引。因此我们用大约1/10的额外存储空间换来了大约全表扫描10倍的定位效率。

03 – 索引的进化

在实际的应用中，这样的定位效率仍然不能满足需求。很多人可能已经想到了，通过排序和查找
算法来减少IO的访问。因此我们开始尝试对Dense Index进行排序存储,并且期望利用排序查

找算法来减少磁盘IO。

- 折半块查找

A. 对Dense Index排序
B. 需要一个数组按顺序存储索引块地址。以块为单位，不存储所有的行的地址。
C. 这个索引块地址数组，也要存储到磁盘上。将其单独存放在一个块链中，如下图所示。
D. 折半查找的时间复杂度是O(log $_2$(*N*))。在上面的列子中，dense索引总共有10,000个块。假设1个块
能存储2000个指针，需要5个块来存储这个数组。通过折半块查找，我们最多只需要读取5（数组块）+ 14（索引块log $_2$(10000)）+1（数据块）=20个块。

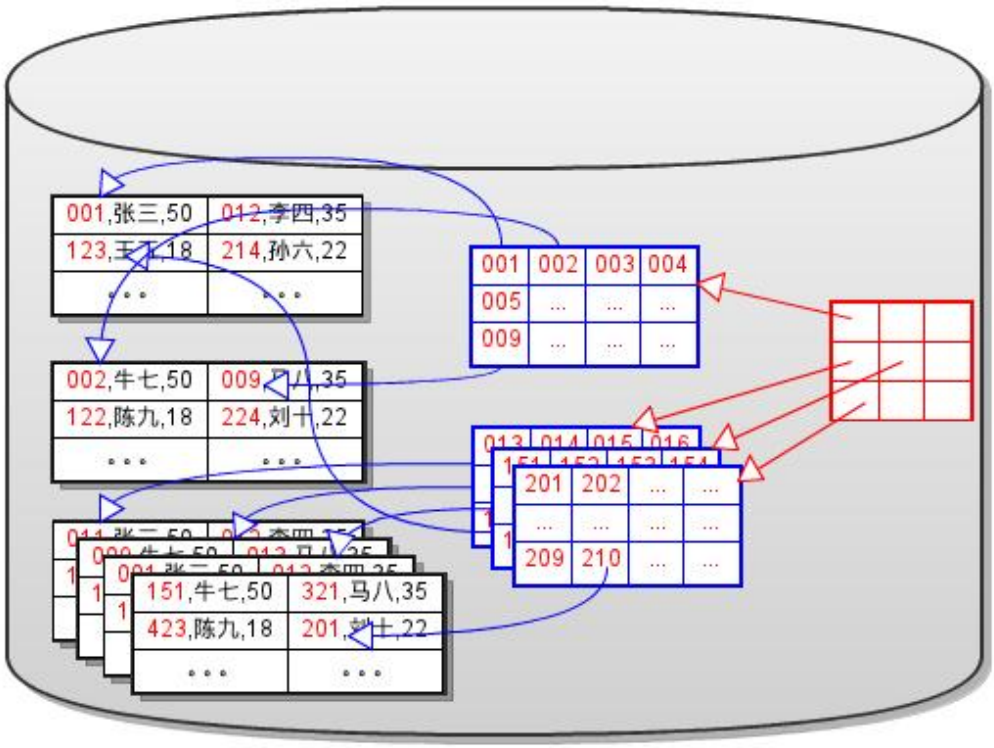
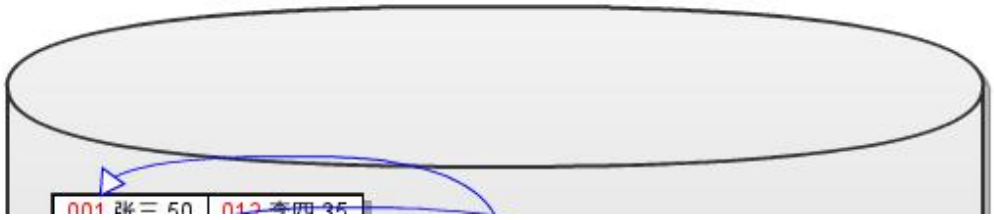


Fig. 4

- Sparse Index

实现基于块的折半查找时发现，读出每个块后只需要和第一行的键值匹配，就可以决定下一个块
的位置（方向）。 因此有效数据是每个块（最后一个块除外）的第一行的数据。还是根据减少无
效数据IO的原则，将每一个块的第一行的数据单独拿出来，和索引数组的地址放到一起。这样就
可以直接在这个数组上进行折半查找了。如下图所示，这个数组就进化成了**Sparse Index**。



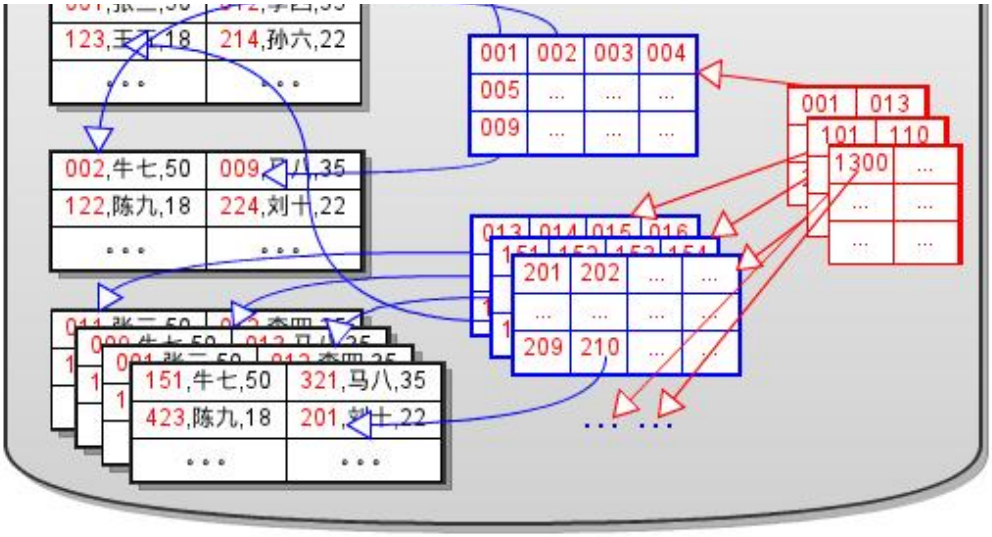


Fig. 5

因为Sparse Index和Dense Index的存储结构是相同的，所以占用的空间也相同。大约需要10个块来存储10000个Dense Index块的地址和首行键值。通过Sparse索引，仅需要读取10(Sparse块)+1(Dense块)+1(数据块)=12个块.

- 多层Sparse Index

因为Sparse Index本身是有序的，所以可以为Sparse Index再建sparse Index。通过这个方法，一层一层的建立 Sparse Indexes,直到最上层的Sparse Index只占用一个块为止,如下图所示.

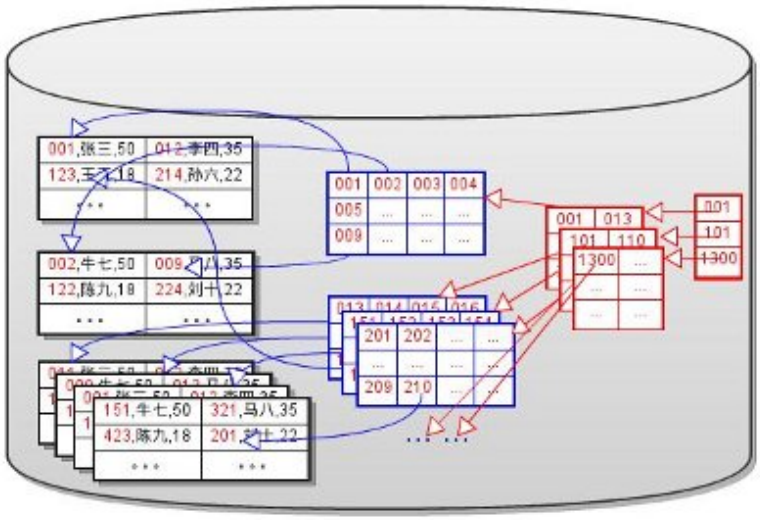


Fig. 6

A. 这个最上层的Sparse Index称作整个索引树的根(root).
B. 每次进行定位操作时，都从根开始查找。
C. 每层索引只需要读出一个块。
D. 最底层的Dense Index或数据称作叶子(leaf).
E. 每次查找都必须要搜索到叶子节点，才能定位到数据。
F. 索引的层数称作索引树的高度(height).
G. 索引的IO性能和索引树的高度密切相关。索引树越高，磁盘IO越多。

在我们的例子中的Sparse Index，只有10个块，因此我们只需要再建立一个Sparse Index.
通过两层Sparse Index和一层Dense Index查找时，只需读取1+1+1+1=4个块。

- Dense Index和Sparse Index的区别

A. Dense Index包含所有数据的键值，但是Sparse Index仅包含部分键值。
Sparse Index占用更少的磁盘空间。
B. Dense Index指向的数据可以是无序的，但是Sparse Index的数据必须是有序的。
C. Sparse Index 可以用来做索引的索引，但是Dense Index不可以。
D. 在数据是有序的时候，Sparse Index更有效。因此Dense Index仅用于无序的数据。
E. 索引扫描(Index Scan)实际上是对Dense Index层进行遍历。

- 簇索引(Clustered Index)和辅助索引(Secondary Index)

如果数据本身是基于某个Key来排序的，那么可以直接在数据上建立sparse索引，
而不需要建立一个dense索引层(可以认为数据就是dense索引层)。 如下图所示：

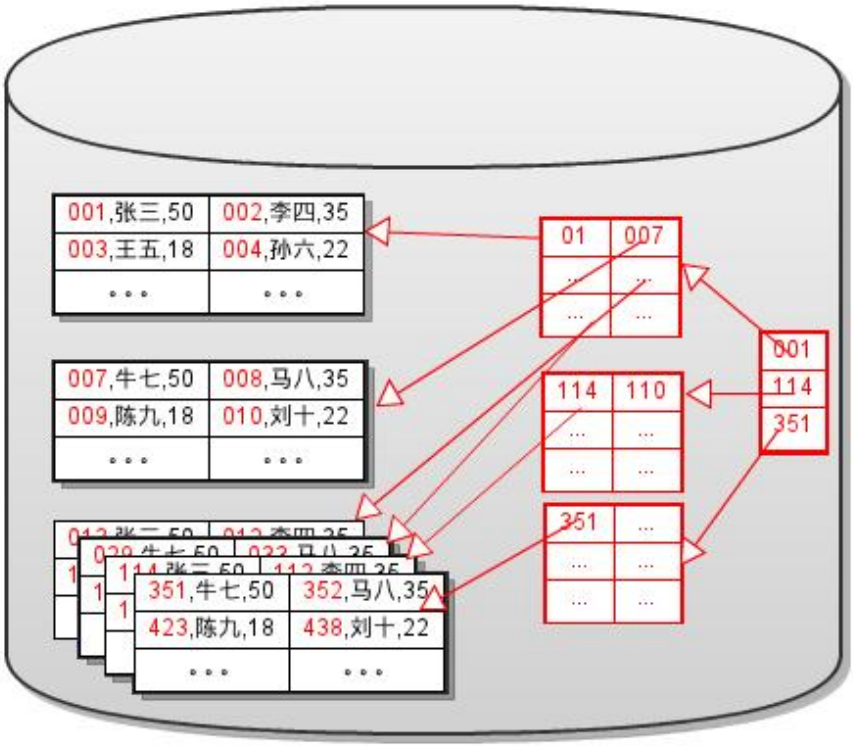


Fig. 7

这个索引就是我们常说的“**Clustered Index**”,而用来排序数据的键叫做主键**Primary Key**.

A. 一个表只能有一个Clustered Index,因为数据只能根据一个键排序.
B. 用其他的键来建立索引树时，必须要先建立一个dense索引层，在dense索引层上对此键的值
进行排序。这样的索引树称作**Secondary Index**.
C. 一个表上可以有多个Secondary Index.
D. 对簇索引进行遍历，实际上就是对数据进行遍历。因此簇索引的遍历效率比辅组索引低。
如SELECT count(*) 操作，使用辅组索引遍历的效率更高。

- 范围搜索(Range Search)

由于键值是有序的，因此可以进行范围查找。只需要将数据块、Dense Index块分别以双向链

表
的方式进行连接， 就可以实现高效的范围查找。如下图所示：

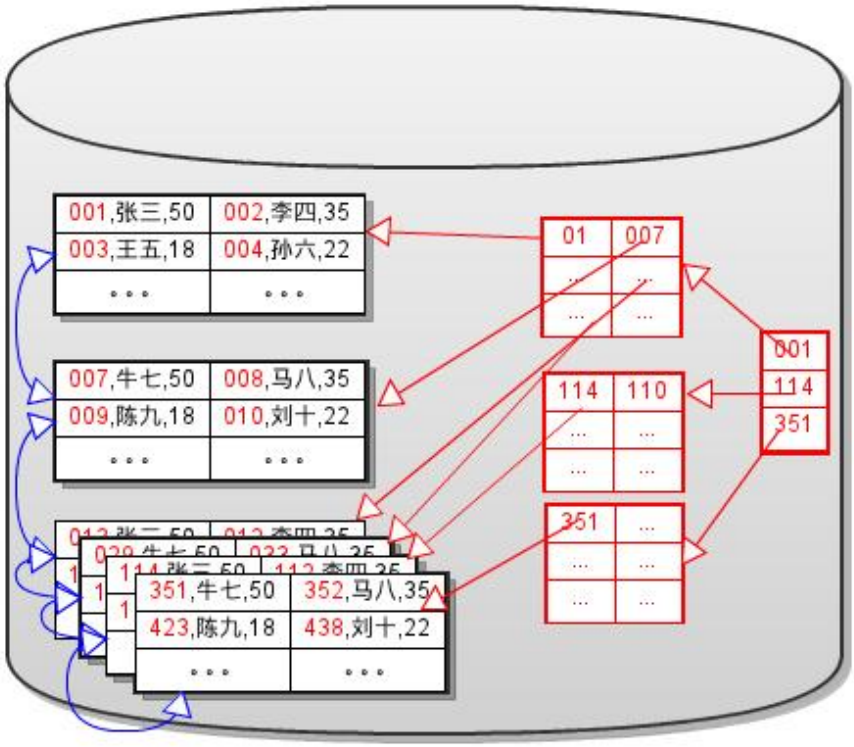


Fig. 8 范围查找的过程： A. 选择一个合适的边界值，定位该值数据所在的块 B. 然后选择合适的方向，在数据块（或Dense Index块）链中进行遍历。C. 直到数据不满足另一个边界值，结束范围查找。 *是不是看着这个索引树很眼熟？换个角度看看这个图吧！*

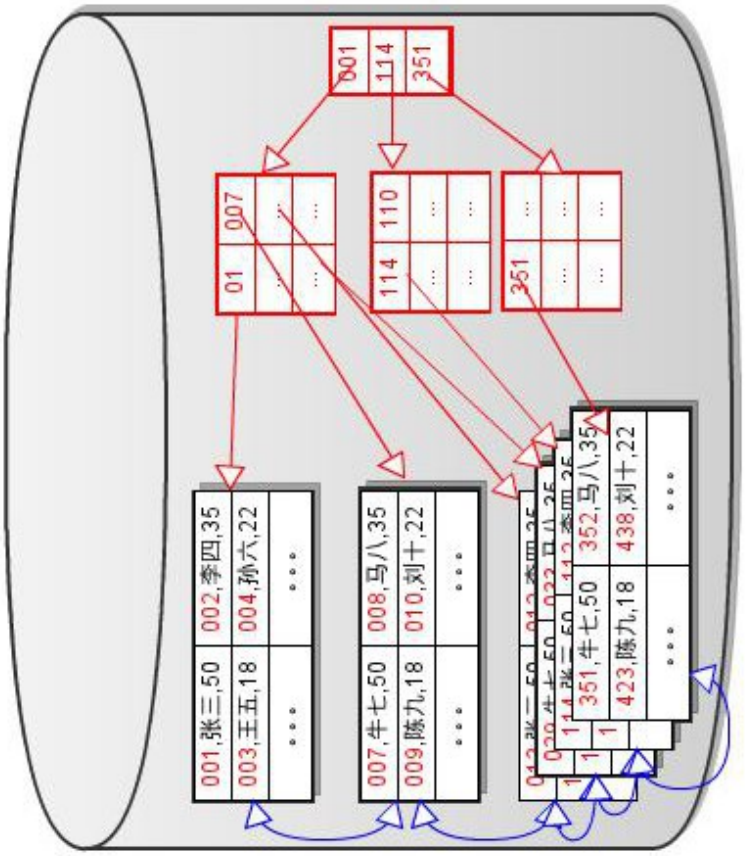


Fig. 9

这分明就是传说中的B+Tree.

- 索引上的操作

A. 插入键值

B. 删除键值

C. 分裂一个节点

D. 合并两个节点

这些操作在教科书上都有介绍，这里就不介绍了。

先写到这吧，实在写不动了，想明白容易，写明白就难了。下一篇里，打算谈谈标准B+Tree的几个问题，以及在

实现过程中，B+Tree的一些变形。

教科书上的B+Tree是一个简化了的，方便于研究和教学的B+Tree。然而在数据库实现时，为了
更好的性能或者降低实现的难度，都会在细节上进行一定的变化。下面以InnoDB为例，来说说这些变化。

04 - Sparse Index中的数据指针

在“由浅入深理解索引的实现(1)”中提到，Sparse Index中的每个键值都有一个指针指向所在的数据页。这样每个B+Tree都有指针指向数据页。如图Fig.10所示：

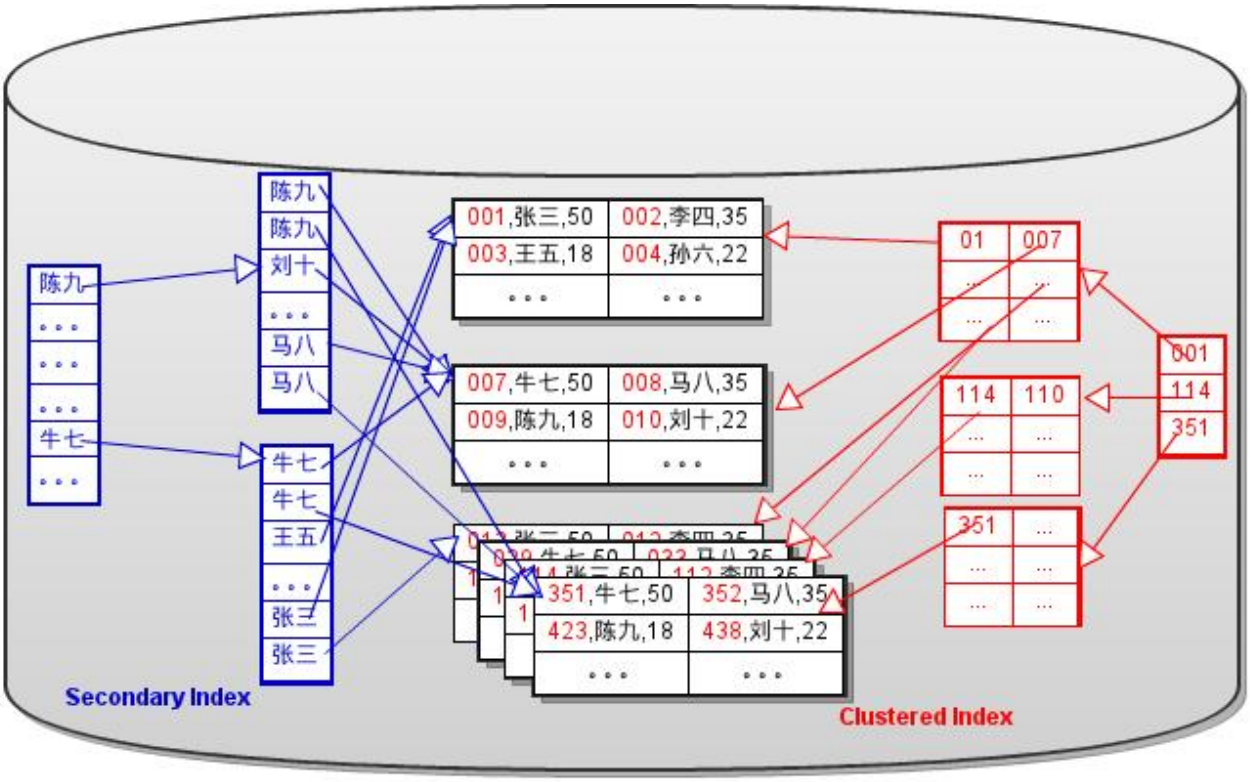


Fig.10

如果数据页进行了拆分或合并操作，那么所有的B+Tree都需要修改相应的页指针。特别是Secondary B+Tree（辅助索引对应的B+Tree), 要对很多个不连续的页进行修改。同时也需要对
这些页加锁，这会降低并发性。

为了降低难度和增加更新(分裂和合并B+Tree节点)的性能，InnoDB 将 Secondary B+Tree中的指针替换成了主键的键值。如图Fig.11所示：

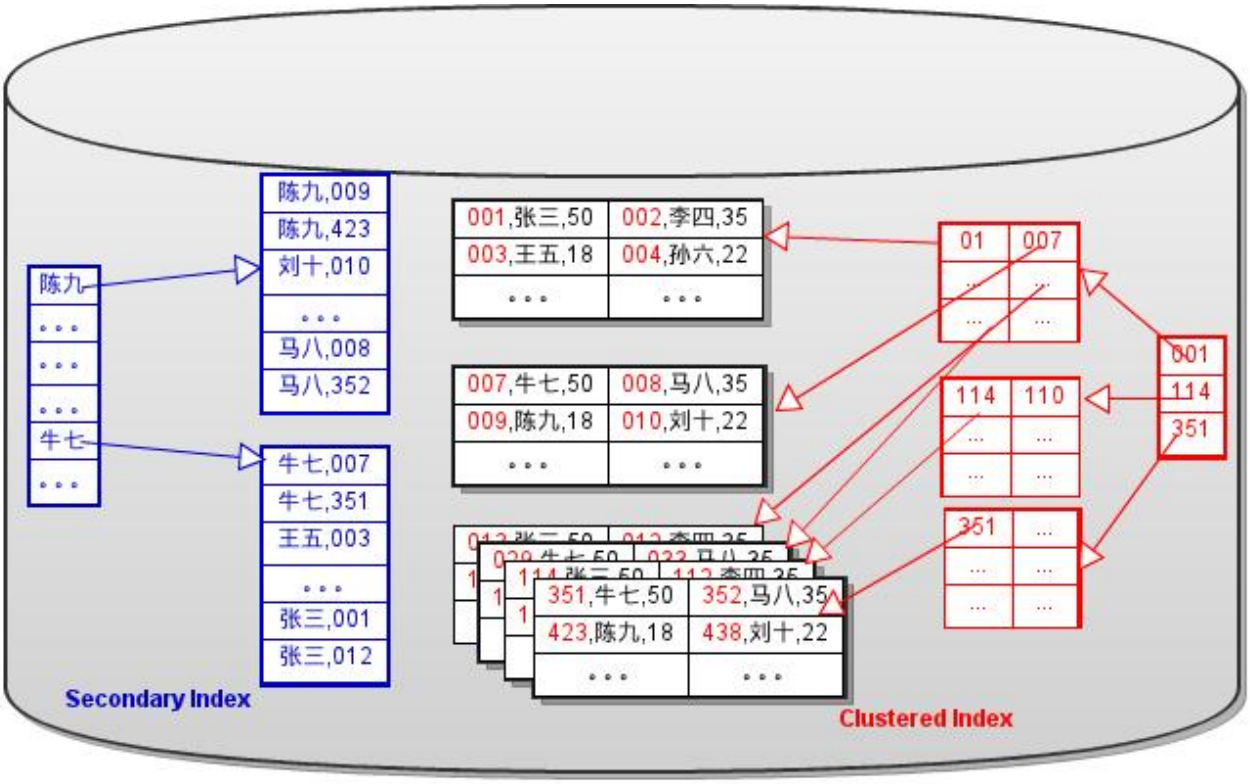


Fig.11

这样就去除了Secondary B+Tree对数据页的依赖，而数据就变成了Clustered B+Tree(簇索引对应的B+Tree)独占的了。对数据页的拆分及合并操作，仅影响Clustered B+Tree. 因此InnoDB的数据文件中存储的实际上就是多个孤立B+Tree。

一个有趣的问题，当用户显式的把主键定义到了二级索引中时，还需要额外的主键来做二级索引的
数据吗(即存储2份主键)? 很显然是不需要的。InnoDB在创建二级索引的时候，会判断主键的字段
是否已经被包含在了要创建的索引中。

接下来看一下数据操作在B+Tree上的基本实现。

- 用主键查询

直接在Clustered B+Tree上查询。

- 用辅助索引查询

A. 在Secondary B+Tree上查询到主键。
B. 用主键在Clustered B+Tree

可以看出，在使用主键值替换页指针后，辅助索引的查询效率降低了。

A. 尽量使用主键来查询数据(索引遍历操作除外).

B. 可以通过缓存来弥补性能，因此所有的键列，都应该尽量的小。

- INSERT

A. 在Clustered B+Tree上插入数据
B. 在所有其他Secondary B+Tree上插入主键。

- DELETE

A. 在Clustered B+Tree上删除数据。
B. 在所有其他Secondary B+Tree上删除主键。

- UPDATE 非键列

A. 在Clustered B+Tree上更新数据。

- UPDATE 主键列

A. 在Clustered B+Tree删除原有的记录(只是标记为DELETED,并不真正删除)。
B. 在Clustered B+Tree插入新的记录。
C. 在每一个Secondary B+Tree上删除原有的数据。(有疑问，看下一节。)
D. 在每一个Secondary B+Tree上插入原有的数据。

- UPDATE 辅助索引的键值

A. 在Clustered B+Tree上更新数据。
B. 在每一个Secondary B+Tree上删除原有的主键。
C. 在每一个Secondary B+Tree上插入原有的主键。

更新键列时，需要更新多个页，效率比较低。

A. 尽量不用对主键列进行UPDATE操作。
B. 更新很多时，尽量少建索引。

05 – 非唯一键索引

教科书上的B+Tree操作，通常都假设”键值是唯一的“。但是在实际的应用中Secondary Index是允
许键值重复的。在极端的情况下，所有的键值都一样，该如何来处理呢?
InnoDB 的 Secondary B+Tree中，主键也是此键的一部分。
Secondary Key = 用户定义的KEY + 主键。如图Fig.12所示:

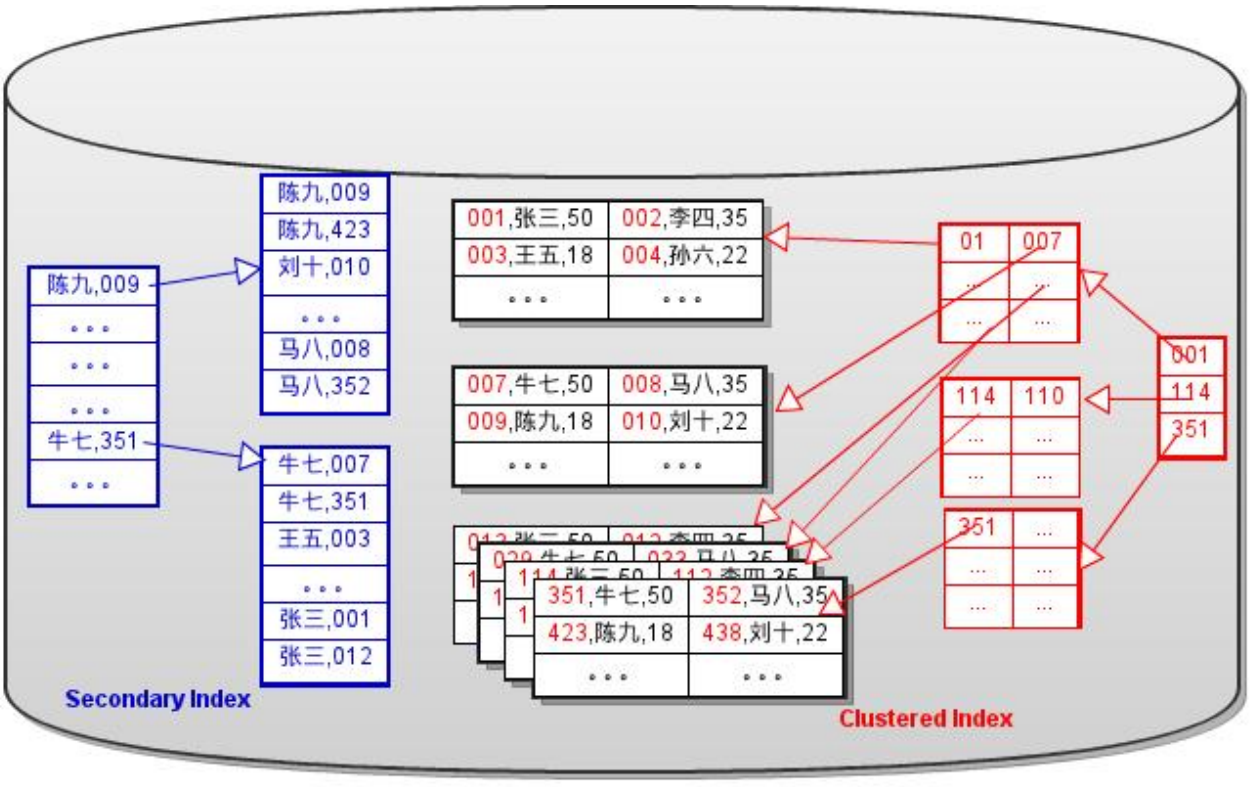


Fig.12

注意主键不仅做为数据出现在叶子节点，同时也作为键的一部分出现非叶子节点。对于非唯一键来说，
因为主键是唯一的，Secondary Key也是唯一的。当然，在插入数据时，还是会根据用户定义的Key，
来判断唯一性。按理说，如果辅助索引是唯一的(并且所有字段不能为空)，就不需要这样做。可是，
InnoDB对所有的Secondary B+Tree都这样创建。

~~*还没弄明白有什么特殊的用途？有知道的朋友可以帮忙解答一下。*~~
~~也许是为了降低代码的复杂性，这是我想到的唯一理由。~~
弄清楚了,即便是非空唯一键，在二级索引的B+Tree中也可能重复，因此必须要将主键加入到非叶子节点。

06 – <Key, Pointer>对

标准的B+Tree的每个节点有K个键值和K+1个指针，指向K+1个子节点。如图Fig.13：

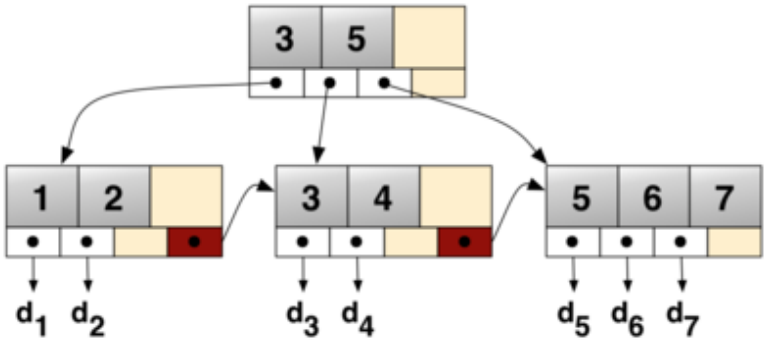


Fig.13(图片来自于WikiPedia)

而在“由浅入深理解索引的实现(1)”中Fig.9的B+Tree上，每个节点有K个键值和K个指针。
InnoDB的B+Tree也是如此。如图Fig.14所示：

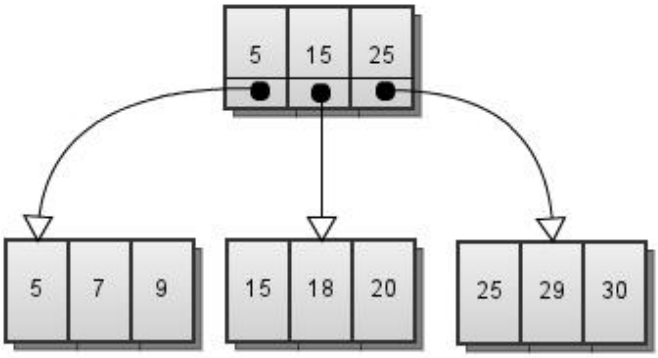


Fig.14

这样做的好处在于，键值和指针一一对应。我们可以将一个<Key,Pointer>对看作一条记录。
这样就可以用数据块的存储格式来存储索引块。因为不需要为索引块定义单独的存储格式，就
降低了实现的难度。

- 插入最小值

当考虑在变形后的B+Tree上进行INSERT操作时,发现了一个有趣的问题。如果插入的数据的健
值比B+Tree的最小键值小时，就无法定位到一个适当的数据块上去(<Key,Pointer>中的Key
代表了子节点上的键值是>=Key的)。例如，在Fig.5的B+Tree中插入键值为0的数据时，无法
定位到任何节点。

在标准的B+Tree上，这样的键值会被定位到最左侧的节点上去。这个做法，对于Fig.5中的
B+Tree也是合理的。Innodb的做法是，将每一层（叶子层除外）的最左侧节点的第一条记录
标
记为最小记录(MIN_REC).在进行定位操作时，任何键值都比标记为MIN_REC的键值大。因
此0
会被插入到最左侧的记录节点上。如Fig.15所示：

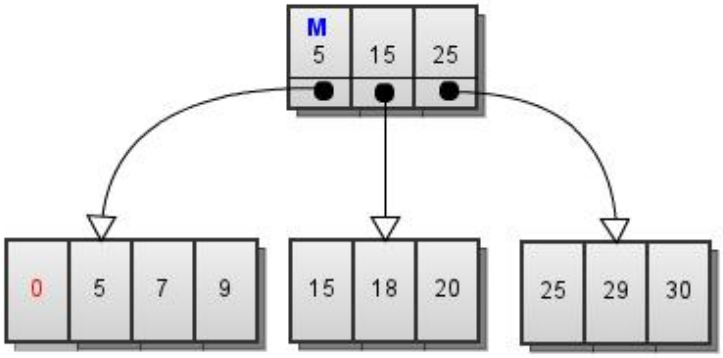


Fig.15

07 – 顺序插入数据

Fig.16是B-Tree的插入和分裂过程，我们看看有没有什么问题?

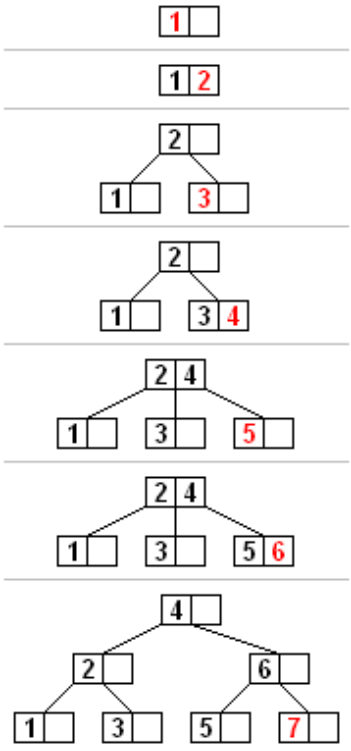


Fig.16(图片来自于WikiPedia)

标准的B-Tree分裂时，将一半的键值和数据移动到新的节点上去。原有节点和新节点都保留一半
的空间，用于以后的插入操作。当按照键值的顺序插入数据时，左侧的节点不可能再有新的数据插入。
因此，会浪费约一半的存储空间。

解决这个问题的基本思路是：分裂顺序插入的B-Tree时，将原有的数据都保留在原有的节点上。
创建一个新的节点，用来存储新的数据。顺序插入时的分裂过程如Fig.17所示：

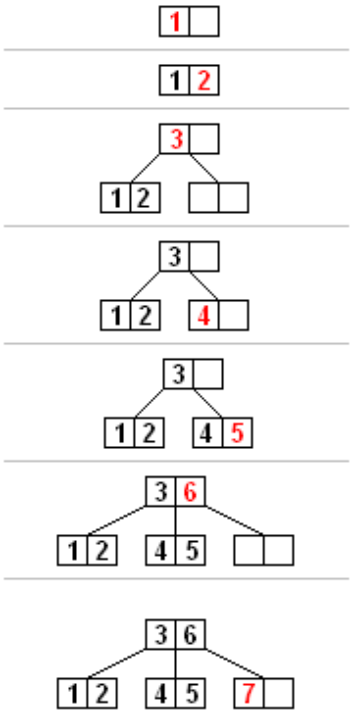


Fig.17

以上是以B-Tree为例，B+Tree的分裂过程类似。InnoDB的实现以这个思路为基础，不过要复杂
一些。因为顺序插入是有方向性的，可能是从小到大，也可能是从大到小的插入数据。所以要区
分不同的情况。如果要了解细节，可参考以下函数的代码。

```
btr_page_split_and_insert();
btr_page_get_split_rec_to_right();
btr_page_get_split_rec_to_right();
```

***InnoDB的代码太复杂了，有时候也不敢肯定自己的理解是对的。因此写了一个小脚本，来打印InnoDB数
据文件中B+Tree。这样可以直观的来观察B+Tree的结构，验证自己的理解是否正确。***
ibd-analyzer.tar

很多知识来自于下面这两本书。“Database Systems: The Complete Book (2nd Edition) ”
“Transaction Processing: Concepts and Techniques”

转载自：http://www.mysqlops.com/2011/11/24/understanding_index.html