【数据结构】B树、B+树详解

B树

前言

首先，为什么要总结B树、B+树的知识呢？最近在学习数据库索引调优相关知识，数据库系统普遍采用B-/+Tree作为索引结构（例如mysql的InnoDB引擎使用的B+树），理解不透彻B树，则无法理解数据库的索引机制；接下来将用最简洁直白的内容来了解B树、B+树的数据结构

另外，B-树，即为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。如人们可能会以为B-树是一种树，而B树又是一种树。而事实上是，B-tree就是指的B树，目前理解B的意思为平衡

B树的出现是为了弥合不同的存储级别之间的访问速度上的巨大差异，实现高效的 I/O。平衡二叉树的查找效率是非常高的，并可以通过降低树的深度来提高查找的效率。但是当数据量非常大，树的存储的元素数量是有限的，这样会导致二叉查找树结构由于树的深度过大而造成磁盘I/O读写过于频繁，进而导致查询效率低下。另外数据量过大会导致内存空间不够容纳平衡二叉树所有结点的情况。B树是解决这个问题的很好的结构

概念

首先，B树不要和二叉树混淆，在计算机科学中，**B树**是一种自平衡树数据结构，它维护有序数据并允许以对数时间进行搜索，顺序访问，插入和删除。B树是二叉搜索树的一般化，因为节点可以有两个以上的子节点。[1]与其他自平衡二进制搜索树不同，B树非常适合读取和写入相对较大的数据块（如光盘）的存储系统。它通常用于数据库和文件系统。

定义

B树是一种平衡的多分树，通常我们说m阶的B树，它必须满足如下条件：

- 每个节点最多只有m个子节点。
- 每个非叶子节点（除了根）具有至少⌈ m/2⌉子节点。
- 如果根不是叶节点，则根至少有两个子节点。
- 具有*k*个子节点的非叶节点包含*k* -1个键。

随笔档案(162)
2020年1月(1)
2019年11月(1)
2019年10月(2)
2019年9月(8)
2019年8月(1)
2019年7月(8)
2019年3月(2)
2018年12月(2)
2018年11月(8)
2018年10月(1)
2018年9月(4)
2018年7月(1)
2018年5月(5)
2018年3月(6)
2017年12月(1)
2017年11月(1)
2017年9月(4)
2017年8月(5)
2017年7月(7)
2017年6月(7)
2017年5月(4)
2017年3月(11)
2017年2月(18)
2017年1月(3)
2016年12月(7)
2016年11月(12)
2016年10月(11)
2016年9月(9)
2016年8月(10)
2016年7月(2)

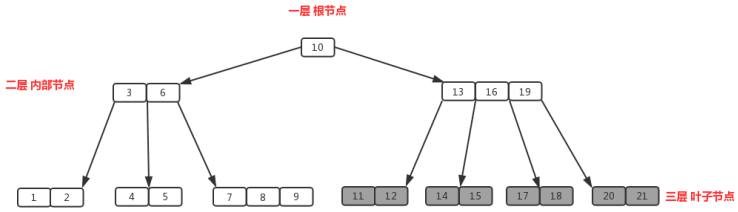
最新评论
1. Re:【数据结构】B树、B+树详解
想问下 为啥网上说B树因为不是有序的所以不支持顺序查找
--asasooo998
2. Re:Python开发【笔记】：git&github 快速入门
好文，顶
--涂小刀
3. Re:Python开发【项目】：选课系统-改良版
@ Richard_Liang非常感谢，如果是原来的[]执行的时候会引发报错list indices must be integers or slices, not str...
--涂小刀

- 所有叶子都出现在同一水平，没有任何信息（高度一致）。

第一次看到这个定义的时候，在想什么鬼？。。。。什么是阶？子节点、飞叶子点、根？？？啥意思！少年别慌。。。

什么是B树的阶？

B树中一个节点的子节点数目的最大值，用m表示，假如最大值为10，则为10阶，如图



所有节点中，节点【13,16,19】拥有的子节点数目最多，四个子节点（灰色节点），所以可以定义上面的图片为4阶B树，现在懂什么是阶了吧

什么是根节点？

节点【10】即为根节点，特征：根节点拥有的子节点数量的上限和内部节点相同，如果根节点不是树中唯一节点的话，至少有俩个子节点（不然就变成单支了）。在m阶B树中（根节点非树中唯一节点），那么有关系式2<= M <=m，M为子节点数量；包含的元素数量 1<= K <=m-1,K为元素数量。

什么是内部节点？

节点【13,16,19】、节点【3,6】都为内部节点，特征：内部节点是除叶子节点和根节点之外的所有节点，拥有父节点和子节点。假定m阶B树的内部节点的子节点数量为M，则一定要符合（m/2）<= M <=m关系式，包含元素数量M-1；包含的元素数量 （m/2）-1<= K <=m-1,K为元素数量。m/2向上取整。

什么是叶子节点?

节点【1,2】、节点【11,12】等最后一层都为叶子节点，叶子节点对元素的数量有相同的限制，但是没有子节点，也没有指向子节点的指针。特征：在m阶B树中叶子节点的元素符合（m/2）-1<= K <=m-1。

好了，概念已经清楚，不用着急背公式， 接着往下看

插入

针对m阶高度h的B树，插入一个元素时，首先在B树中是否存在，如果不存在，即在叶子结点处结束，然后在叶子结点中插入该新的元素。

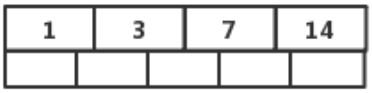
- 若该节点元素个数小于m-1，直接插入；

- 若该节点元素个数等于m-1，引起节点分裂；以该节点中间元素为分界，取中间元素（偶数个数，中间两个随机选取）插入到父节点中；
- 重复上面动作，直到所有节点符合B树的规则；最坏的情况一直分裂到根节点，生成新的根节点，高度增加1；

上面三段话为插入动作的核心，接下来以5阶B树为例，详细讲解插入的动作；

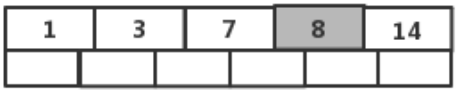
5阶B树关键点:

- 2<=根节点子节点个数<=5
- 3<=内节点子节点个数<=5
- 1<=根节点元素个数<=4
- 2<=非根节点元素个数<=4



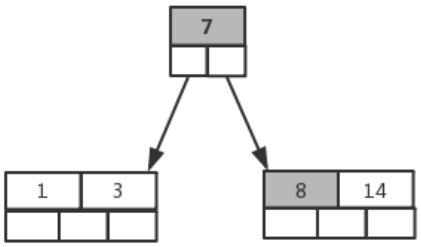
（1）

插入8



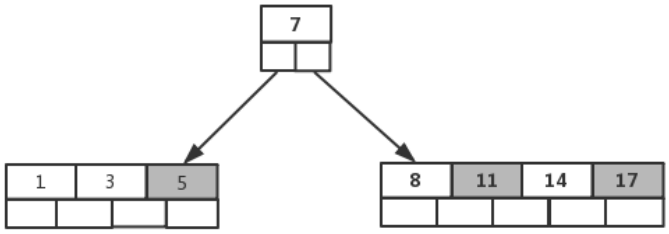
（2）

图（1）插入元素【8】后变为图（2），此时根节点元素个数为5，不符合 1<=根节点元素个数<=4，进行分裂（真实情况是先分裂，然后插入元素，这里是为了直观而先插入元素，下面的操作都一样，不再赘述），取节点中间元素【7】，加入到父节点，左右分裂为2个节点，如图（3）



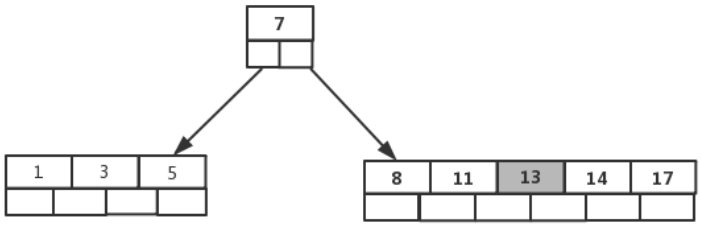
（3）

接着插入元素【5】，【11】，【17】时，不需要任何分裂操作，如图（4）



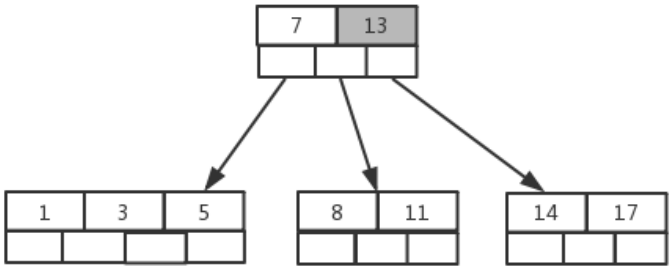
（4）

插入元素【13】



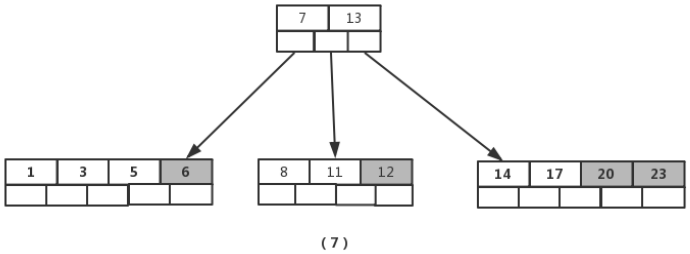
（5）

节点元素超出最大数量，进行分裂，提取中间元素【13】，插入到父节点当中，如图（6）



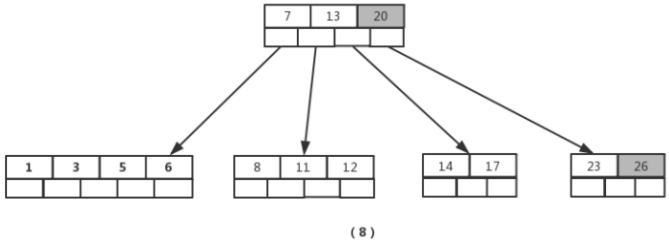
（6）

接着插入元素【6】，【12】，【20】，【23】时，不需要任何分裂操作，如图（7）



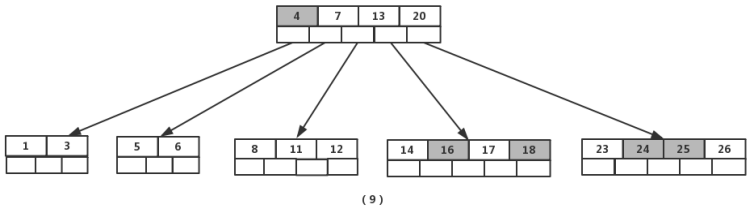
（7）

插入【26】时，最右的叶子结点空间满了，需要进行分裂操作，中间元素【20】上移到父节点中，注意通过上移中间元素，树最终还是保持平衡，分裂结果的结点存在2个关键字元素。



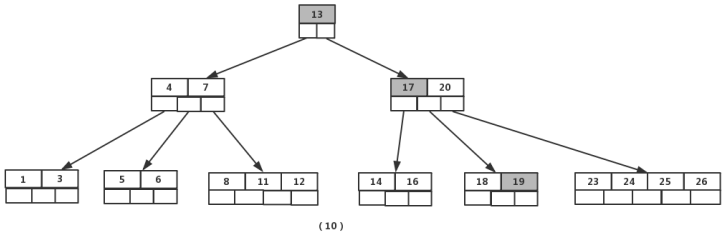
（8）

插入【4】时，导致最左边的叶子结点被分裂，【4】恰好也是中间元素，上移到父节点中，然后元素【16】,【18】,【24】,【25】陆续插入不需要任何分裂操作



（9）

最后，当插入【19】时，含有【14】,【16】,【17】,【18】的结点需要分裂，把中间元素【17】上移到父节点中，但是情况来了，父节点中空间已经满了，所以也要进行分裂，将父节点中的中间元素【13】上移到新形成的根结点中，这样具体插入操作的完成。



（10）

删除

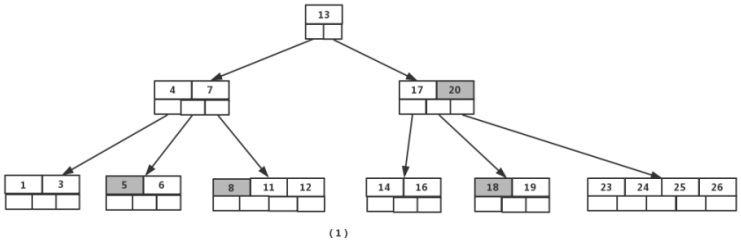
首先查找B树中需删除的元素,如果该元素在B树中存在，则将该元素在其结点中进行删除；删除该元素后，首先判断该元素是否有左右孩子结点，如果有，则上移孩子结点中的某相近元素(“左孩子最右边的节点”或“右孩子最左边的节点”)到父节点中，然后是移动之后的情况；如果没有，直接删除。

- 某结点中元素数目小于（m/2）-1,(m/2)向上取整，则需要看其某相邻兄弟结点是否丰满；
- 如果丰满（结点中元素个数大于(m/2)-1），则向父节点借一个元素来满足条件；
- 如果其相邻兄弟都不丰满，即其结点数目等于(m/2)-1，则该结点与其相邻的某一兄弟结点进行“合并”成一个结点；

接下来还以5阶B树为例，详细讲解删除的动作；

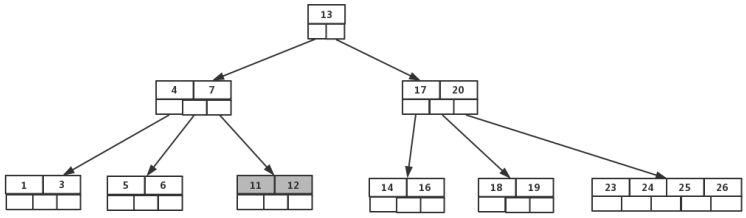
- 关键要领，元素个数小于 2（m/2 -1）就合并，大于4（m-1）就分裂

如图依次删除依次删除【8】,【20】,【18】,【5】



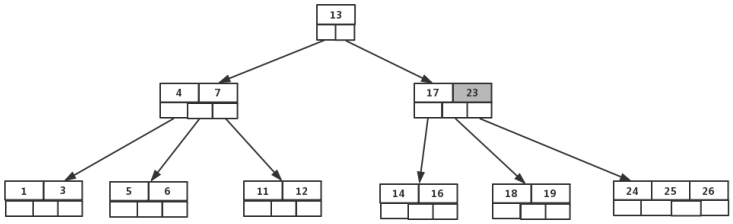
(1)

首先删除元素【8】，当然首先查找【8】，【8】在一个叶子结点中，删除后该叶子结点元素个数为2，符合B树规则，操作很简单，咱们只需要移动【11】至原来【8】的位置，移动【12】至【11】的位置（也就是结点中删除元素后面的元素向前移动）



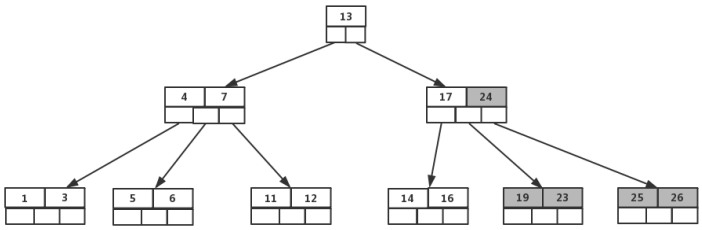
(2)

下一步，删除【20】,因为【20】没有在叶子结点中，而是在中间结点中找到，咱们发现他的继承者【23】(字母升序的下个元素)，将【23】上移到【20】的位置，然后将孩子结点中的【23】进行删除，这里恰好删除后，该孩子结点中元素个数大于2，无需进行合并操作。



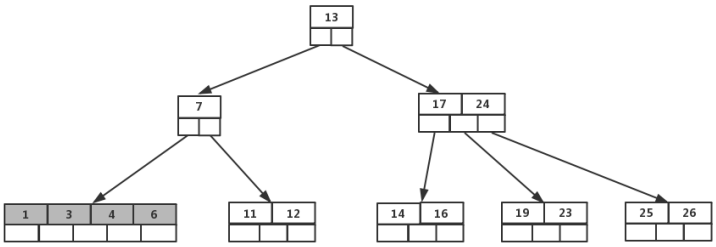
(3)

下一步删除【18】，【18】在叶子结点中,但是该结点中元素数目为2，删除导致只有1个元素，已经小于最小元素数目2,而由前面我们已经知道：如果其某个相邻兄弟结点中比较丰满（元素个数大于ceil(5/2)-1=2），则可以向父结点借一个元素，然后将最丰满的相邻兄弟结点中上移最后或最前一个元素到父节点中，在这个实例中，右相邻兄弟结点中比较丰满（3个元素大于2），所以先向父节点借一个元素【23】下移到该叶子结点中，代替原来【19】的位置，【19】前移；然【24】在相邻右兄弟结点中上移到父结点中，最后在相邻右兄弟结点中删除【24】，后面元素前移。



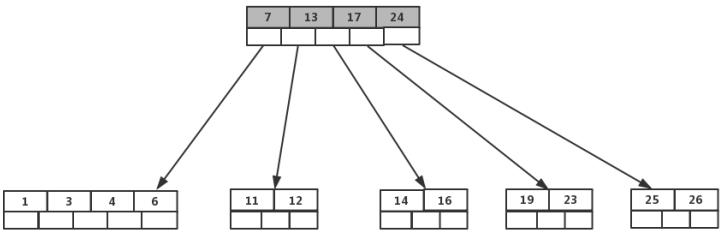
（4）

最后一步删除【5】， 删除后会导致很多问题，因为【5】所在的结点数目刚好达标，刚好满足最小元素个数（ceil(5/2)-1=2）,而相邻的兄弟结点也是同样的情况，删除一个元素都不能满足条件，所以需要该节点与某相邻兄弟结点进行合并操作；首先移动父结点中的元素（该元素在两个需要合并的两个结点元素之间）下移到其子结点中，然后将这两个结点进行合并成一个结点。所以在该实例中，咱们首先将父节点中的元素【4】下移到已经删除【5】而只有【6】的结点中，然后将含有【4】和【6】的结点和含有【1】,【3】的相邻兄弟结点进行合并成一个结点。



（5）

也许你认为这样删除操作已经结束了，其实不然，在看看上图，对于这种特殊情况，你立即会发现父节点只包含一个元素【7】，没达标（因为非根节点包括叶子结点的元素K必须满足于2=<K<=4，而此处的K=1），这是不能够接受的。如果这个问题结点的相邻兄弟比较丰满，则可以向父结点借一个元素。而此时兄弟节点元素刚好为2，刚刚满足，只能进行合并，而根结点中的唯一元素【13】下移到子结点，这样，树的高度减少一层。



（6）

看完插入，删除，想必也把B树的特征掌握了，下面普及下其他知识，换个脑子

磁盘IO与预读

计算机存储设备一般分为两种：内存储器(main memory)和外存储器(external memory)。

内存储器为内存，内存存取速度快，但容量小，价格昂贵，而且不能长期保存数据(在不通电情况下数据会消失)。

外存储器即为磁盘读取，磁盘读取数据靠的是机械运动，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，寻道时间指的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。下图是计算机硬件延迟的对比图，供大家参考：

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

考虑到磁盘IO是非常高昂的操作，计算机操作系统做了一些优化，当一次IO时，不光把当前磁盘地址的数据，而是把相邻的数据也都读取到内存缓冲区内，因为局部预读性原理告诉我们，当计算机访问一个地址的数据的时候，与其相邻的数据也会很快被访问到。每一次IO读取的数据我们称之为一页(page)。具体一页有多大数据跟操作系统有关，一般为4k或8k，也就是我们读取一页内的数据时候，实际上才发生了一次IO，这个理论对于索引的数据结构设计非常有帮助。

事实1： 不同容量的存储器，访问速度差异悬殊。

- 磁盘(ms级别) << 内存(ns级别)， 100000倍
- 若内存访问需要1s，则一次外存访问需要一天
- 为了避免1次外存访问，宁愿访问内存100次...所以将最常用的数据存储在最快的存储器中

事实2： 从磁盘中读 1 B，与读写 1KB 的时间成本几乎一样

从以上数据中可以总结出一个道理，索引查询的数据主要受限于硬盘的I/O速度，查询I/O次数越少，速度越快，所以B树的结构才应需求

而生；B树的每个节点的元素可以视为一次I/O读取，树的高度表示最多的I/O次数，在相同数量的总元素个数下，每个节点的元素个数越多，高度越低，查询所需的I/O次数越少；假设，一次硬盘一次I/O数据为8K，索引用int(4字节)类型数据建立，理论上一个节点最多可以为2000个元素，2000*2000*2000=8000000000，80亿条的数据只需3次I/O（理论值），可想而知，B树做为索引的查询效率有多高；

另外也可以看出同样的总元素个数，查询效率和树的高度密切相关

B树的高度

一棵含有N个总关键字数的m阶的B树的最大高度是多少?

log（m/2）(N+1)/2 + 1 ，log以（m/2）为低，(N+1)/2的对数再加1

算法如下

B 树的高度

B 树上大部分操作所需的磁盘存取次数与 B 树的高度成正比。下面来分析 B 树的最坏情况高度。

定理 18.1 如果 $n \geqslant 1$，则对任意一棵包含 n 个关键字、高度为 h、最小度数 $t \geqslant 2$ 的 B 树 T，有：

$$h \leqslant \log_t \frac{n+1}{2}$$

证明： 如果一棵 B 树的高度为 h，其根结点包含至少一个关键字而其他结点包含至少 $t-1$ 个关键字。这样，在深度 1 至少有两个结点，在深度 2 至少有 $2t$ 个结点，在深度 3 至少有 $2t^2$ 个结点，等等，直到深度 h 至少有 $2t^{h-1}$ 个结点。图 18-4 给出了 $h=3$ 时的一棵树。因此，关键字的个数 n 满足不等式

439

$$n \geqslant 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right) = 2t^h - 1$$

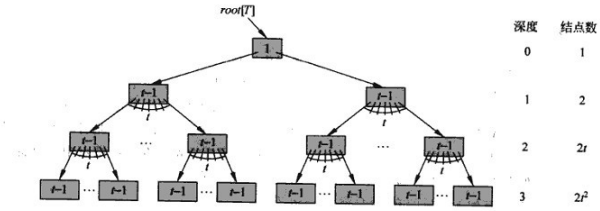


图 18-4 一棵高度为 3 的 B 树，它包含最小可能的关键字数。在每个结点 x 内显示的是 $n[x]$

⊖ B* 树是 B 树的另一种常用变形，在这种树中，要求每个内结点至少为 2/3 满，而不是像 B 树要求的至少半满。

B+树

B+树是应文件系统所需而产生的B树的变形树，那么可能一定会想到，既然有了B树，又出一个B+树，那B+树必然是有很多优点的

B+树的特征：

- 有m个子树的中间节点包含有m个元素（B树中是k-1个元素），每个元素不保存数据，只用来索引；
- 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）；

- 所有的非终端结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 树的非终节点也包含需要查找的有效信息）；

为什么说B+树比B树更适合数据库索引?

1）B+树的磁盘读写代价更低

B+树的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了；

2）B+树查询效率更加稳定

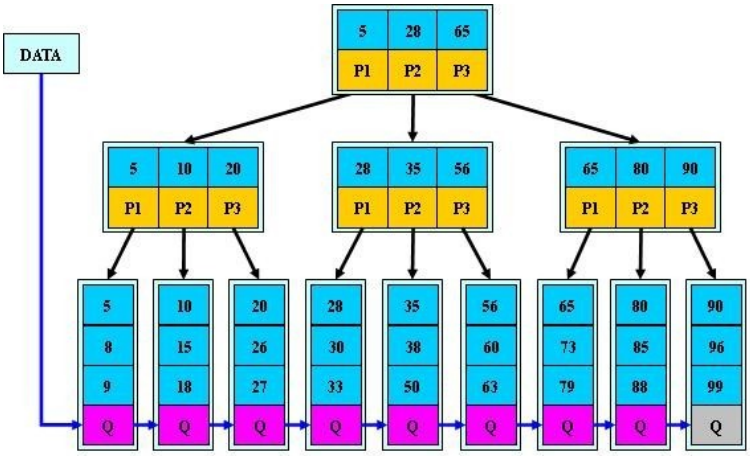
由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当；

3）B+树便于范围查询（最重要的原因，范围查找是数据库的常态）

B树在提高了IO性能的同时并没有解决元素遍历的我效率低下的问题，正是为了解决这个问题，B+树应用而生。B+树只需要去遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树不支持这样的操作或者说效率太低；不懂可以看看这篇解读-》范围查找

补充：B树的范围查找用的是中序遍历，而B+树用的是在链表上遍历；

B+树如下：



分类：数据结构

Assassinの
关注 - 10
粉丝 - 359
+加关注

« 上一篇： Python开发【源码剖析】 Dict对象
» 下一篇： 按月分表存储过程

posted @ 2019-07-28 18:51 Assassinの 阅读(2937) 评论(4)
编辑 收藏

评论列表

#1楼 2019-08-20 20:15 头上有多云

能搞懂这篇文章，也得掉一层皮啊。哈哈哈慢慢看

支持(0) 反对(0)

#2楼 2019-11-18 13:01 段涛。

请问一下，我看到文中对B树的插入操作，在最后插入【19】时，插入之前是5阶B树，插入之后变成了3阶B树。所以插入操作是有可能引起树的降阶是吗?

支持(0) 反对(0)

#3楼 [楼主] 2019-11-20 12:00 Assassinの

@ 段涛。
并没有降阶，B树定义的时候就是5阶，只需要符合下面公式就还是5阶
3<=内节点子节点个数<=5

支持(0) 反对(0)

#4楼 2020-02-12 17:27 asasooo998

想问下 为啥网上说B树因为不是有序的 所以不支持顺序查找

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部