B树、B+树、LSM树以及其典型应用场景 - CSDN博客

笔记本:	data-structure
创建时间:	2018/8/24 11:11
标签:	lsm-tree
URL:	https://blog.csdn.net/u010853261/article/details/78217823

转 B树、B+树、LSM树以及其典型应用场景

前言

动态查找树主要有：二叉查找树、平衡二叉树、红黑树、B树、B+树。前面三种是典型的二叉查找树，查找的时间复杂度是O(log2N)与树的深度有关系，那么降低树的深度也就可以提升查找效率。这时就提出了平衡多路查找树，也就是B树以及B+树。

B树和B+树非常典型的场景就是用于关系型数据库的索引(MySQL)

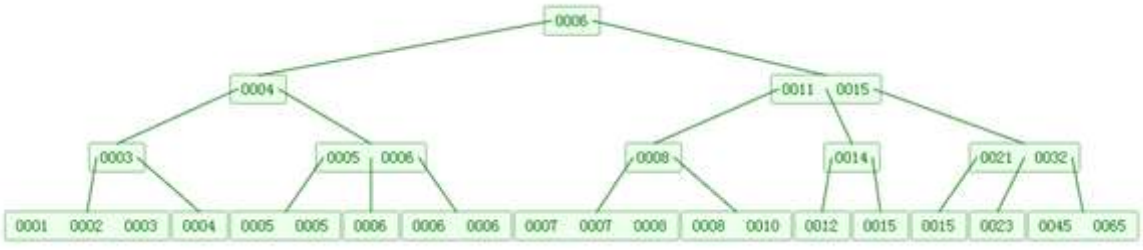
B树

B树是一种平衡多路搜索树，B树与红黑树最大的不同在于，B树的结点可以有多个子女，从几个到几千个。那为什么又说B树与红黑树很相似呢?因为与红黑树一样，一棵含n个结点的B树的高度也为O（lgn），但可能比一棵红黑树的高度小许多，应为它的分支因子比较大。所以，B树可以在O（lgn）时间内，实现各种如插入（insert），删除（delete）等动态集合操作。

B树的定义如下：

- 根节点至少有两个子节点
- 每个节点有M-1个key，并且以升序排列
- 位于M-1和M key的子节点的值位于M-1 和M key对应的Value之间
- 其它节点至少有M/2个子节点
- 所有叶子结点位于同一层；

下图是一个M=4的4阶的B树：



B树的搜索：从根结点开始，对结点内的关键字（有序）序列进行二分查找，如果命中则结束，否则进入查询关键字所属范围的儿子结点；重复，直到所对应的儿子指针为空，或已经是叶子结点；

B树的特性：

1. 关键字集合分布在整颗树中；
2. 任何一个关键字出现且只出现在一个结点中；
3. 搜索有可能在非叶子结点结束(树中所有结点都存储数据，与B+树这一点不同)；
4. 其搜索性能等价于在关键字全集内做一次二分查找；

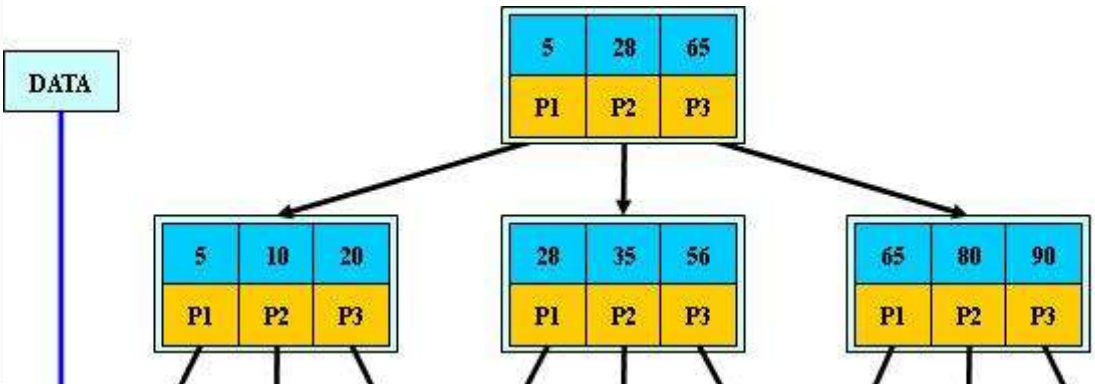
下面是一个B树插入的演示动画，依次插入：
6 10 4 14 5 11 15 3 2 12 1 7 8 8 6 3 6 21 5 15 15 6 32 23 45 65 7 8 6 5 4

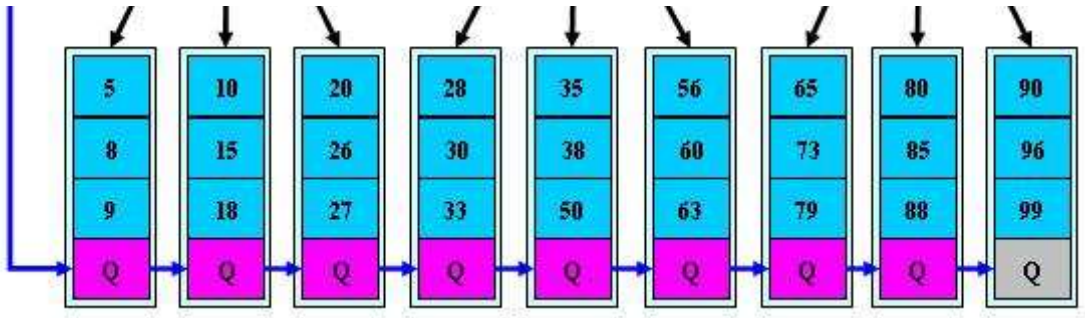
B+树

B+树是对B树的一种变形，与B树的差异在于：

1. 有n棵子树的结点中含有n个关键字，每个关键字不保存数据，只用来索引，所有数据都保存在叶子节点。
2. 所有的叶子结点中包含了全部关键字的信息，及指向含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顺序链接。
3. 所有的非终端结点可以看成是索引部分，结点中仅含其子树（根结点）中的最大（或最小）关键字。
4. 为所有叶子结点增加一个链指针，便于区间查找和遍历。
5. 所有关键字都在叶子结点出现；

如下图一个M=3 的B+树：





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+的特性：

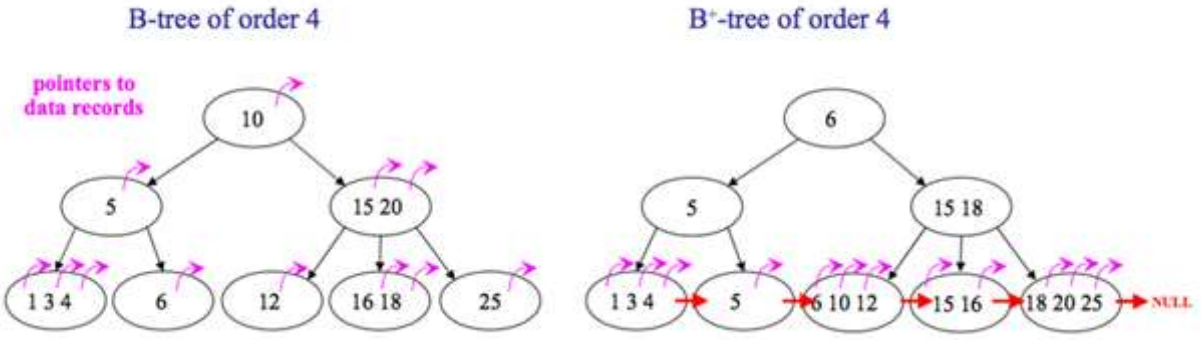
1. 非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层；
2. B+树的叶子结点都是相链的，因此对整棵树的遍历只需要一次线性遍历叶子结点即可。而且由于数据顺序排列并且相连，所以便于区间查找和搜索。而B树则需要进行每一层的递归遍历。相邻的元素可能在内存中不相邻，所以缓存命中性没有B+树好。

B树和B+树总结：

B树：多路搜索树，每个结点存储M/2到M个关键字，非叶子结点存储指向关键字范围的子结点；所有关键字在整颗树中出现，且只出现一次，非叶子结点可以命中；

B+树：在B-树基础上，为叶子结点增加链表指针，所有关键字都在叶子结点中出现，非叶子结点作为叶子结点的索引；B+树总是到叶子结点才命中；

B+树虽然优点很多，但是B树也有优点，其优点在于，由于B树的每一个节点都包含key和value，因此经常访问的元素可能离根节点更近，因此访问也更迅速。下面是B 树和B+树的区别图：



为什么说B+tree比B树更适合实际应用中操作系统的文件索引和数据库索引?

（1) B+tree的磁盘读写代价更低

B+tree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。

举个例子，假设磁盘中的一个盘块容纳16bytes，而一个关键字2bytes，一个关键字具体信息指针2bytes。一棵9阶B-tree(一个结点最多8个关键字)的内部结点需要2个盘快。而B+ 树内部结点只需要1个盘快。当需要把内部结点读入内存中的时候，B 树就比B+ 树多一次盘块查找时间(在磁盘中就是盘片旋转的时间)。

（2）B+tree的查询效率更加稳定
由于非叶子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树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题。正是为了解决这个问题，B+树应运而生。B+树只要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）。

LSM树

目前常见的主要的三种存储引擎是：哈希、B+树、LSM树：

- 哈希存储引擎：是哈希表的持久化实现，支持增、删、改以及随机读取操作，但不支持顺序扫描，对应的存储系统为key-value存储系统。对于key-value的插入以及查询，哈希表的复杂度都是O(1)，明显比树的操作O(n)快,如果不需要有序的遍历数据，哈希表性能最好。
- B+树存储引擎是B+树的持久化实现，不仅支持单条记录的增、删、读、改操作，还支持顺序扫描（B+树的叶子节点之间的指针），对应的存储系统就是关系数据库（Mysql等）。
- LSM树（Log-Structured MergeTree）存储引擎和B+树存储引擎一样，同样支持增、删、读、改、顺序扫描操作。而且通过批量存储技术规避磁盘随机写入问题。当然凡事有利有弊，LSM树和B+树相比，LSM树牺牲了部分读性能，用来大幅提高写性能。

上面三种引擎中，LSM树存储引擎的代表数据库就是HBase.

LSM树核心思想的核心就是放弃部分读能力，换取写入的最大化能力。LSM Tree ，这个概念就是结构化合并树的意思，它的核心思路其实非常简单，就是假定内存足够大，因此不需要每次有数据更新就必须将数据写入到磁盘中，而可以先将最新的数据驻留在内存中，等到积累到足够多之后，再使用归并排序的方式将内存内的数据合并追加到磁盘队尾(因为所有待排序的树都是有序的，可以通过合并排序的方式快速合并到一起)。

日志结构的合并树（LSM-tree）是一种基于硬盘的数据结构，与B+tree相比，能显著地减少硬盘磁盘臂的开销，并能在较长的时间提供对文件的高速插入（删除）。**然而LSM-tree在某些情况下，特别是在查询需要快速响应时性能不佳。**通常LSM-tree适用于索引插入比检索更频繁的应用系统。

LSM树和B+树的差异主要在于读性能和写性能进行权衡。在牺牲的同时寻找其余补救方案：

（a）**LSM具有批量特性，存储延迟。**当写读比例很大的时候（写比读多），LSM树相比于B树有更好的性能。因为随着insert操作，为了维护B+树结构，节点分裂。读磁盘的随机读写概率会变大，性能会逐渐减弱。

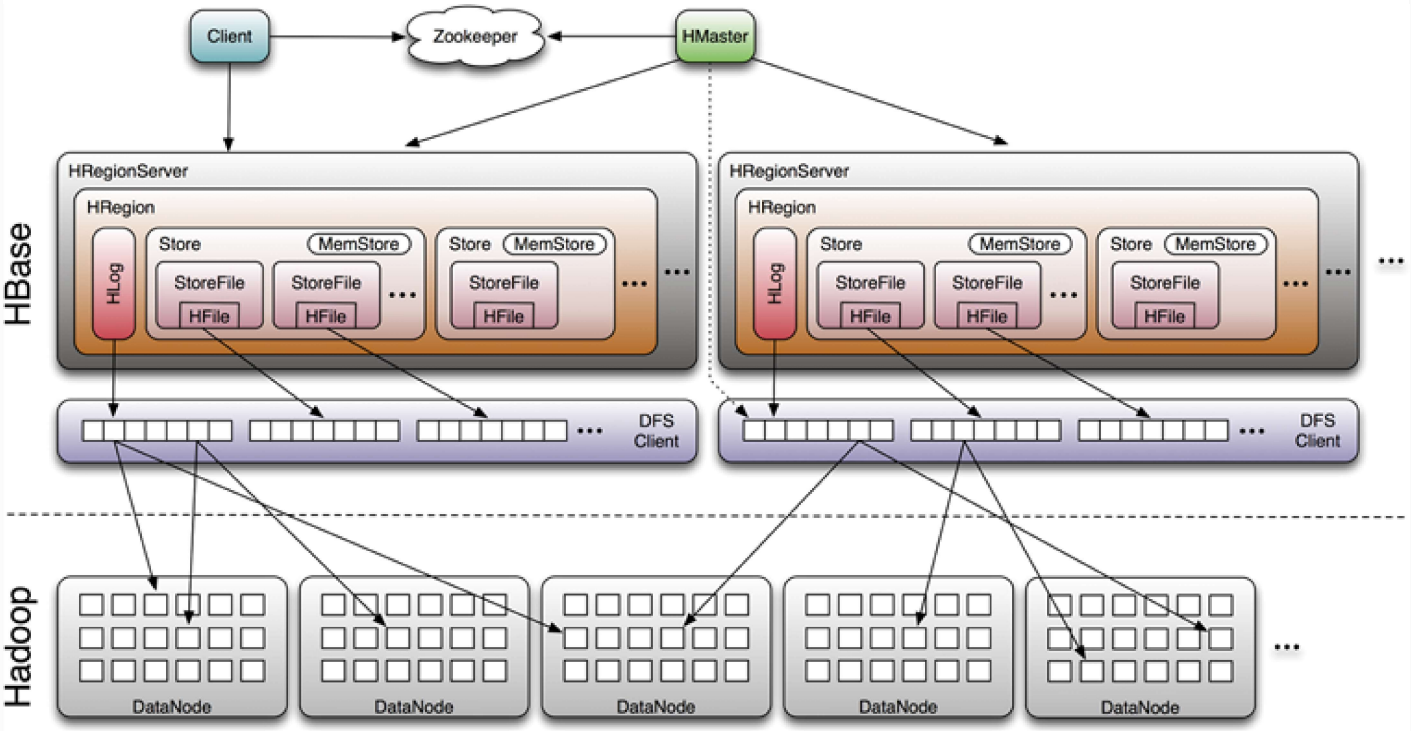
（b）**B树的写入过程**：对B树的写入过程是一次原位写入的过程，主要分为两个部分，首先是查找到对应的块的位置，然后将新数据写入到刚才查找到的数据块中，然后再查找到块所对应的磁盘物理位置，将数据写入去。当然，在内存比较充足的时候，因为B树的一部分可以被缓存在内存中，所以查找块的过程有一定概率可以在内存内完成，不过为了表述清晰，我们就假定内存很小，只够存一个B树块大小的数据吧。可以看到，在上面的模式中，需要两次随机寻道（一次查找，一次原位写），才能够完成一次数据的写入，代价还是很高的。

（c）**LSM优化方式**：

1. Bloom filter: 就是个带随机概率的bitmap,可以快速的告诉你，某一个小的有序结构里有没有指定的那个数据的。于是就可以不用二分查找，而只需简单的计算几次就能知道数据是否在某个小集合里啦。效率得到了提升，但付出的是空间代价。
2. compact:小树合并为大树:因为小树性能有问题，所以要有个进程不断地将小树合并到大树上，这样大部分的老数据查询也可以直接使用log2N的方式找到，不需要再进行(N/m)*log2n的查询了

Hbase中存储设计主要思想

SML树原理把一棵大树拆分成N棵小树，它首先写入内存中，随着小树越来越大，内存中的小树会flush到磁盘中，磁盘中的树定期可以做merge操作，合并成一棵大树，以优化读性能。



以上这些大概就是HBase存储的设计主要思想，这里分别对应说明下：

- 因为小树先写到内存中，为了防止内存数据丢失，写内存的同时需要暂时持久化到磁盘，对应了HBase的Me

mStore和HLog

- MemStore上的树达到一定大小之后，需要flush到HRegion磁盘中（一般是Hadoop DataNode），这样MemStore就变成了DataNode上的磁盘文件StoreFile，定期HRegionServer对DataNode的数据做merge操作，彻底删除无效空间，多棵小树在这个时机合并成大树，来增强读性能。