RUTGERS, THE STATE UNIVERSITY OF NEW JERSEY

The Introduction and Analysis to B-Trees

Xinyu Lyu (xl422)

xl422@scarletmail.rutegrs.edu

Course Name: DATA STRUCT & ALGS

Date: 3rd Apr

Abstract:

The paper will talk about the data structure and some basic operations of the B-tree. Also, we give some analysis of the search, insert cost. Finally, the paper introduces some variations about B-Trees and their applications, like the B+ tree and the B* tree.

I. Introduction

Nowadays, with the development of computing and internet, a considerable amount of information becomes accessible to us. Therefore, efficiently accessing the target information is of immense significance. Without appropriate searching algorithm, it is by no means possible for us to develop the modern world based on the computational infrastructure. In recent years, with the appearance of the web applications, more and more amount of data is available for us to access. Meanwhile, we are meeting a new challenge to access them efficiently.

Thus, in this paper, we want to introduce an extension data structure and algorithm of the balanced-tree called B Trees. It could support the external search of the data stored in the disk or on the web, which is beyond the storage ability of the computer memory. Moreover, over the years, the software system is gradually trending towards claudication which is more convenient for sharing, computing and storing. Moreover, the difference between the concepts of local and web files are fading away. Therefore, with B-trees, we could do the searching and insert of the symbol table with billions of data by only using 4-5 references

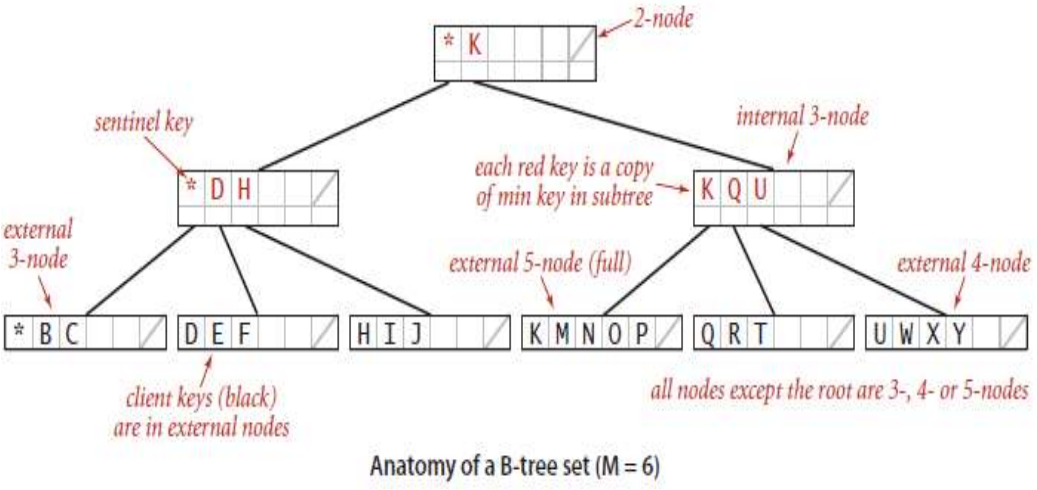
II. Definitions

1. Cost-models:

Before describing the B-trees algorithm, we need to find out its cost models. We should find out a simple and basic data storage model to express it due to the various storage mechanicals. As mentioned on algs4, we could use the term page to describe the adjacent data blocks and the term probe for accessing one page. Moreover, we reasonably assume that we need to read the page into local memory from the web, then it will be more effective for later data obtaining. Also, when taking advantage of the B-trees, we can minimize the number of times for page access. Therefore, as defined on algs4, the cost models for B-tree is the count of the page access for external searching.

2. B-Trees structure:

B-Tree is an extension of the 2-3 tree. However, unlike the 2-3 trees, B-Trees do not store the data in the tree nodes. The internal nodes are all the copies of the smallest keys in its child nodes. Moreover, every copy associated with a link. Therefore, we can easily separate the index and the contents, which are convenient for data accessing, like referring to the index in the dictionary. Moreover, we set both the upper bound and the lower bound for the number of the key-link pairs for each node. Apart from the root node, every node has at least M/2 pairs and at most M-1 key-link pairs (M is often an even figure). For the root node, it must have at most M-1 pairs of key-link, and no less than two pairs. As mentioned in algs4, we could use B-tree of order M to specify the value of M. For example, for a B-tree of order 6, every node has at least three but no more than five key-link pairs. For the root node, it may have no less than 2 and at most M-1 key-link pairs.



B-Trees structure

Like the diagram above, the keys in the nodes are all in the order designed for searching. Moreover, there are two kinds of nodes: the first kind of node is the internal node which stores the smallest copies of keys from its child nodes. Moreover, the second kind of node is the external nodes which are the leaf nodes at the bottom of the B-trees. The external nodes store the references to the actual data. Furthermore, every key link to its child node. Also, the keys in its linked child nodes are bigger than the key in the corresponding key-link pair. Moreover, they are smaller than the next key in their father node. For instance, for the node contains only K, Q and U keys, the keys in the left linked child node K, M, N, O, P are all bigger than the key K in its corresponding key-link pair. Also, they are all smaller than the next key Q in its father's next key-link pair. It is the same as the Q, R, T, and U, W, X, Y keys in the other key-link pairs. Last, when initializing the root node, we set a Sentinel key (*) which is the smallest key in the whole B-trees. The Sentinel will help simplify the implementation codes.

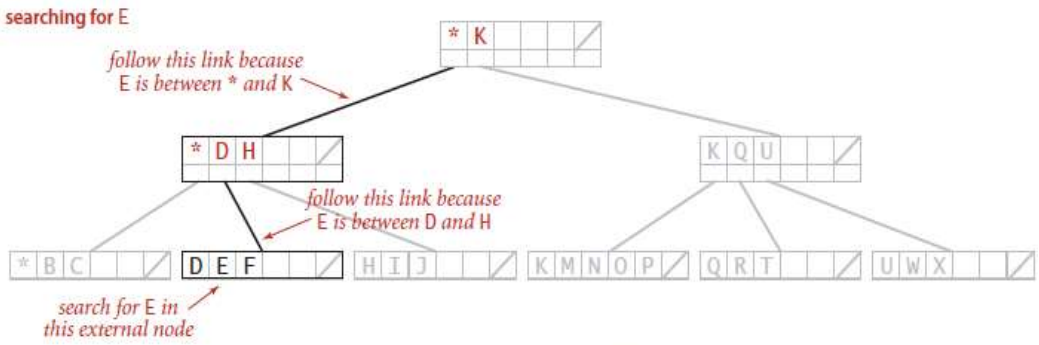
3. Operations:

Search:

The search operation is to find the page with the specific keys. The search operation always starts with the root node. After compared the target key with the keys in the internal nodes, we could find the appropriate key-link pair, based on the characteristic of B-tree, for further search. Then recursively searching the target keys and it will always end up with visiting the external nodes (leaf nodes at the bottom). If we find the corresponding key in the external nodes, we terminate the process with a search hit. If there isn't the target key, return search miss.

Peruse code:

```
If the current page is in the external node,
        If find return true
        If not find return false
Else if the current page is in the internal node
        Recursive to search in the subtree which probably has the target key
```

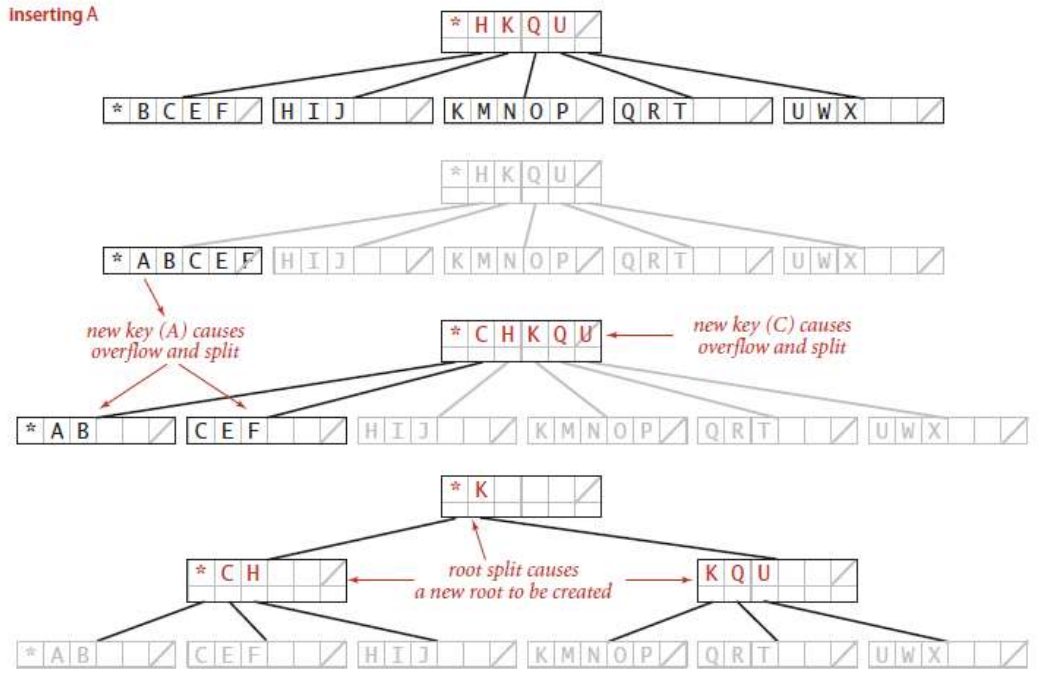


Searching in a B-tree set (M = 6)

More specifically, for the B-tree of order six above, it has 3-nodes with 3-links, 5-nodes with 5-links and 2-nodes with 2-links. When searching for E, we first find E is less than the K but bigger than *(the smallest key in the whole b-tree), then we search the left key-link pairs to its child nodes. After that, we may find key E in the external node, then return search hit.

Insert:

Another pressing operation is the insertion. Just like the 2-3 tree, at first, we reclusively search the target key to find out whether it exists at the bottom pages. If it exists, return a search hit. Otherwise, we add it to the bottom external nodes in ordered position. There is a case that after insert if the node has already been over-full which is a 6-node more than 5(M-1, M=6), we should split it into two 3-nodes with a 2-node father node. Therefore, the full node could only temperately exist, then we should replace anyone over-full 6-nodes linked to k-nodes father with two 3-nodes linked to (k+1)-nodes. The following picture is a simple example of node-splitting when inserting the node A.



Inserting a new key into a B-tree set

Peruse code:

```
Initialize x as root.
While x is not leaf until x is the leaf.
    Find the child of x that is going to be traversed next. Let the child be y.
    If y is not full, change x to point to y.
    If y is full, split it and change x to point to one of the two parts of y.
        If k is smaller than mid key in y, then set x as first part of y.
        Else set x as second part of y. When we split y, we move a key from y to its parent x.
Simply insert k to x.
```

Therefore, we could come with the characteristics that the B-trees is wholly balanced and every node has 2 to M-1 link-key pairs. For the root node, it could have at least two key-link pairs.
As the object to store the actual external data of the B-tree, it is necessary to import the Page API to realize some operations to Page object to implement constructing the B tree.

Delete:

Since delete is very complicated, we need to consider too much care about it. We list all the possible cases below without describing operations in detail.

Case 1:

If the key to be deleted is already in a leaf node (at the bottom), and we remove it not just because of leaf node having too few keys. Then we can remove that key. Key k is in node x, and x is a leaf, merely delete k from x.

Case 2:

If key k is in an internal node x, we need to consider such three cases:

Case 2a:

If the child y that precedes k in node x has at least t keys, then find the predecessor key k1 in the subtree rooted at y. Recursively delete k1 and replace k with k1 in x

Case 2b:

If the child z that follows k in node x has at least t keys, find the successor k1, delete and replace as mentioned before.

Case 2c:

If both y and z have only t−1 keys, merge k and z into y. Therefore, both k and the link to z are removed from x. Also, y now contains 2t−1 keys, then k is deleted.

Case 3:

If key k does not appear in an internal node x, the root of the proper subtree must contain k. If the root has t−1 keys, there are two cases to be executed below. At last, recursively to the appropriate child of x.

Case 3a:

If the root has only t−1 keys but has a sibling with t keys, give the root an extra key by moving a key from x to the root, moving a key from the roots immediate left or right sibling up into x, and moving the appropriate child from the sibling to x.

Case 3b:

If the root and all its siblings have t−1 keys, we should merge the root with one of its siblings. It will involve moving a key down from x into the new merged node. Then it becomes the median key for that node.

III. Analysis

The cost of the search and insert operations for the B-Tree of order M is between and$^{\log_{M/2}N}$. For B-Tree, it increases its height by insert at the bottom, then splits and creates the new page at the top of the tree. Moreover, every node has at most M-1 key-link pairs and at least M/2 pairs. Therefore, when all the pages have M-1 key-link pairs, the B-tree achieves its shortest height as $\log_M N$. And the cost for searching and inserting reaches the minimal values about $\log_M N$. In the worst case, all the pages have M/2 key-link pairs. Because it is completely balanced, the height achieves its maximal value as$^{\log_{M/2}N}$and the cost for both searching and inserting is $\log_M N$.

Also, there is a little improvement that we could save the root node in the memory to reduce the times of probe by 1, for the root node is the most frequently accessed node.

Moreover, according to A.Yao, he proved that the average number of key-link pairs is Mln2 for B-tree, so the wasted space accounts for 44%.

IV. Applications

B-Trees are mostly used for organizing data index and database index on disk files, for instance B-Tree Index are use in Mysql to speed up the data access.

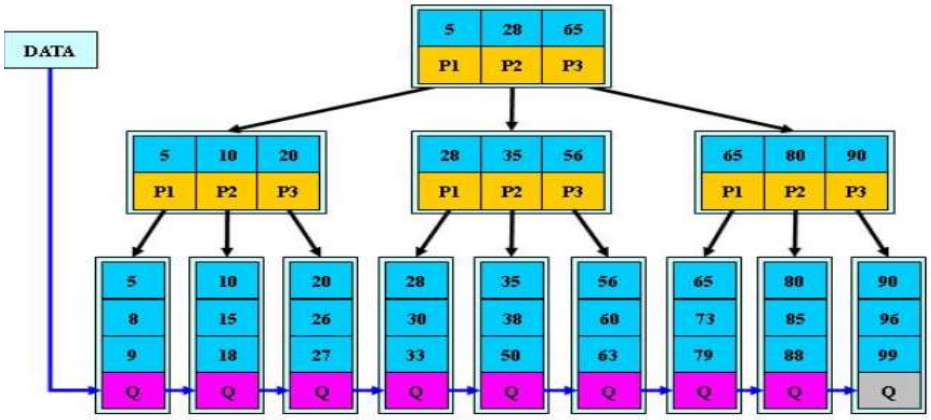
V. Related Work

There are many variations for the B-tree. One idea of the variations is to store as many internal nodes in the B-tree as possible. It will make the b-tree flatter the tree and save the time for search and insert. The other variations are to merge the brother nodes before splitting into new nodes to improve the usage efficiency. Among such variations, B+ trees, B* trees R-trees and Finger Tree are most broadly used.

B+ trees:

Unlike B-trees, B+ trees only store the keys in the leaves which means the leaves store all the keys and corresponding pages. So, the brother nodes will not get involved with the split in B+ trees, for it only generates a new node and assigns it with the half keys from the original node. Then add the reference to the new node.

Moreover, B+ trees are more appropriate for the index of production file system and productions databases in the actual OS. Because the cost of the disks I/O operations are much lower and the search efficiency more stable than B-tree.

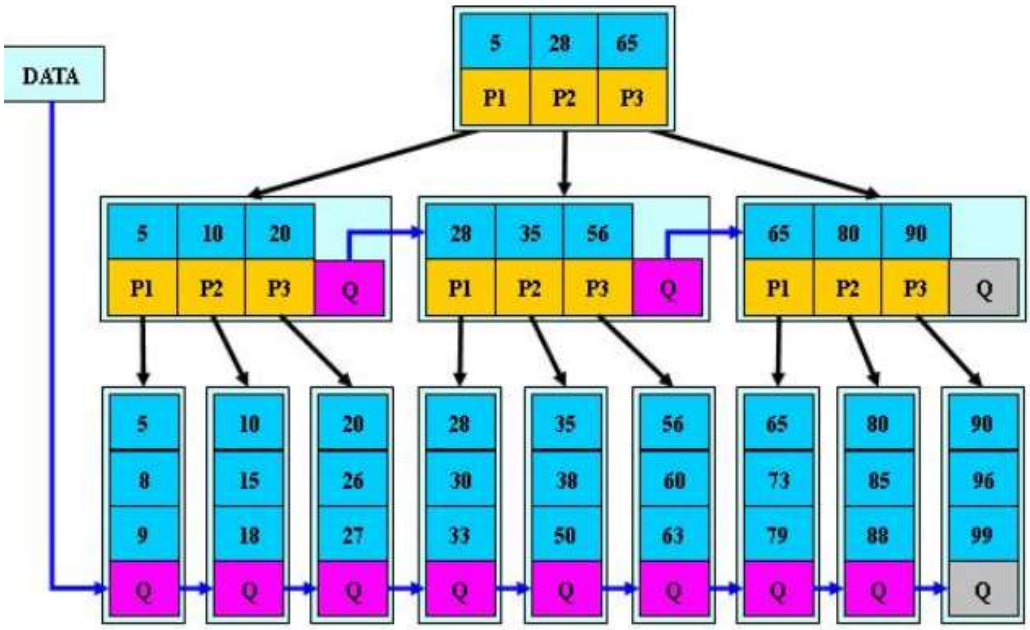


Structures for B+trees

B* trees:

For B* tree, it additionally adds the pointers from Non-leaf nodes and Non-root nodes to the brother nodes based on B-trees.

Unlike the B-tree, the split of B* tree is a little different from the former one. When one node is full, if the next brother node is not full, move a part of data to the brother node. Then insert the key into the original node. Finally, modify the keys in the father node, since the key range in the brother node has been changed. In another case, if the brother node is also full, we need to generate a new node between the original node and the brother one. Then both extract 1/3 keys into the new node and add a new pointer pointing to the new node in the father node. Therefore, the B*tree can make better use of the space than the B+ trees.



Structures for B*Trees

VI. Conclusions

The paper tells about the data structure and some basic operations of the B-tree. Also, we give some analysis of the search, insert with logs-log cost. Then the paper introduces some variations about B-Trees and their applications, like the B+ tree and the B* tree. Based on all above, we can see that B- trees are efficient when doing the search or I/O operations of the enormous file system. It could optimize for system reading and write with large blocks of data. It reduces the time for the process of locating the files, based on the index, to accelerate the file access. Such data structure can describe the external memory. Moreover, nowadays, B-trees has been widely used in databases and filesystems. Moreover, it will do much more significant benefits to the development of web services and file OS in the future.

References:

[1] Acta Informatica 1, 290---306 (1972) x9 by Sprmger-Verlag 1972 Symmetric Binary B-Trees" Data Structure and Maintenance Algorithms* R. Bayer Received January 24, 1972

[2] Acta Informatica 9, 1-21 (1977) 9 by Springer-Verlag 1977 Concurrency of Operations on B-Trees R. Bayer* and M. Schkolnick IBM Research Laboratory, San Jos6, CA 95193, USA

[3] Efficient Locking for Concurrent Operations on B-Trees PHILIP L. LEHMAN Carnegie-Mellon University and S. BING YAO Purdue University

[4] AL OF COMPUTER AND SYSIEM SCIENCES 33, 275-296 (1986) Concurrent Operations on B*-Trees with Overtaking YEHOSHUA SAGIV* Department of Computer Science, Hebrew University, Givat Ram 91904, Jerusalem, Israel Received August 20, 1985; revised June 16, 1986

[5] Multi-Disk B-trees* Bernhard Seegertand Per-~ke Larson Dept. of Computer Science, University of Wate SIGMOD '91 Proceedings of the 1991 ACM SIGMOD international conference on Management of data Pages 436-445