CS618: Group 14
Caching in B+-tree

Sumedh Masulkar
Student Id : 40
Roll Number : 11736
sumedh@iitk.ac.in
Dept. of CSE

Milind Solanki
Student Id : 26
Roll Number : 11422
milind@iitk.ac.in
Dept. of CSE

Banothu Raj Kumar
Student Id : 48
Roll Number : 14111007
rajb@iitk.ac.in
Dept. of CSE

Indian Institute of Technology, Kanpur

Final report
27th April, 2015

Abstract

B+ tree is probably the most widely implemented indexing structure as it provides very efficient search and update operations. We have implemented the Cache Coherent B+ tree (CCB+tree), a variant of B+ tree which can reduce the query time as compared to B+ tree. The CCB+tree makes use of the unused space in the internal nodes of a B+tree to cache frequently queried leaf node addresses, thus saving node accesses. Experiments show that the CCB+tree can outperform the traditional B+tree where the query size is huge and certain queries are much more popular than the others.

1 Introduction and Problem Statement

The B+-tree is an elegant structure for indexing ordered data. In [1], the authors have proposed a technique to improve the performance of B+-tree, on the basis of an observation that the space of internal nodes on an average is only used around 67%. A variant was implemented called Cache coherent B+-tree(CCB+-tree), which improves the performance of the search queries, by utilizing the unused space in the internal nodes. In the internal nodes the pointers to the popular leaf nodes are cached. Suppose a popular point is cached in the root node, then for the query this node, we need not access any other node as we can directly reach that node. This saves a lot of random I/O cost. The popularity of a node is maintained after queries, and more popular nodes are cached closer to the root node. This reduces the workload when some queries are repeated or are similar.

1.1 Related Material

We have followed approach poposed in [1] throughout the project. The data sets and queries were generated randomly with different popular points and different frequencies of these popular points. Each query file had 5000 queries and data files had 10^6 entries.

2 Algorithm or Approach

2.1 Node Structure

The structure of a CCB+-tree is quite similar to B+-tree, and also have added cache entries(in internal nodes) and popularity information(in leaf nodes), which is the count of the times the node has been visited. In the leaf nodes, we store the number of it has been read, which we call the popularity count of the node. The structures of internal node and leaf node are better explained below:
An internal node consists of two lists L_1 and L_2, where L_1 is a list of pointers and keys $(p_0, K_1, p_1, K_2, p_2, ..., K_s, p_s)$; as in $B^+ - tree$ of order M, where $M = 2a1$ and $a \leq s \leq M$. If $s = M$, $L2$ is an empty list; otherwise $L2$ is a list of elements of the form (c_i, K_i, p_i, K_i), where $r \leq (M - s)/2$; p_i is a pointer to cached leaf node; K_i and Ki are the smallest and largest keys in that leaf node pointed by p_i; c_i is a popularity count. These elements are cache entries. List L_2 can contain at most $(M - s)/2$ elements. When $L2$ is

non-empty and has $(M - s)/2$ cache entries, to insert a new point in the L_1 we have to delete the least popular entry of L_2 to accommodate the new entry. Leaf nodes have the form (C_t, K_i, p_i, p_x), where C_t is a popularity count; p_x is a pointer to the next leaf node; Ki, pi are the standard leaf node entries.

2.2 Cache entry

Cache entries are stored backwards, starting from the end of the page to ensure efficient utilization of space, and for easy updation and deletion. Internal nodes one level above the leaf nodes do not store cache entries, since this will never save any node accesses. In this implementation only leaf nodes are cached within its own sub-tree. This makes the cache entries more relevant to the searches.

2.3 How is it maintained

When searching, a parent stack is maintained to keep track of all nodes visited while searching for a point.

- When a leaf node is visited, its popularity count is updated. If a cache entry for this node already exists in internal nodes above this node, the entry is updated. If any ancestor node has empty space or some cache entry with lower popularity count than this node, the cache information for this node is inserted. The ancestor nodes are present in the parent stack.

- If an internal node is visited, the node is inserted into parent stack to update later if needed. Then, the cache entries of this node are searched. Since, the entries contain range as well as pointer, if the range contains the search point, the leaf node is directly accessed, thus the nodes below this node need not be accessed which saves lots of random I/Os. Also, we can see if a node(internal) is not visited, it will not be present in the parent stack, thus removing the need to update the nodes below this. Thus, the topmost node which contains this leaf node, will be updated, and its parents if needed which is exactly what should be done. Even if the entry for that node gets removed from this node later, then we would go below this internal node, and thus get the topmost parent node again which contains the leaf node.

2.

Algorithm 1 AddNewCache($leafnode$,$path$)

```
1:  path ← head of path;
2:  currentNode ← node with address path.offset;
3:  while(currentNode is at cache level)
4:      if(cache space is available)
5:          cache leafnode in currentNode
6:          return;
7:      elseif(currentNode is at the lowest cache level
8:        AND popularity of leafnode >the least popular
9:        ity in currentNode)
10:         remove the cache entry with least popularity;
11:         cache leafnode in currentNode;
12:         return;
13:     elseif(popularity of leafnode >the least
14:       popularity in currentNode)
15:         InsertCache(currentNode,leafnodeinfo);
16:         return;
17:     else
18:         path ← next of path;
19:         currentNode ← node with address
20:           path.offset;
21: End AddNewCache
```

3 Results

For experiments, we created two instances of CCB+ tree, one which adds popular points into the internal nodes when inserting the data too (CCB+(1)), and another which only maintains the cache points only when the queries are being run CCB+(2), since we are mainly interested in reducing the query times. Then, we run different query files on these programs, varying number of popular points and frequencies of these points to see the difference. (All time results are in milliseconds.) All the results are for point search. Other searches follow a similar trend.

	B+-tree	CCB+Tree(1)	CCB+Tree(2)
min	0	0	0
max	10	10	10
mean	0.08	0.31	0.4
std.deviation	0.89	1.73	1.9

Table 1: For uniform distribution of data and query points

In Table.2, $pxfy$ in the first column, here x is the number of popular points and y is the frequency of each popular point in the query file.

Algorithm 2 InsertCache(node,leafnode)

$outCacheEntry \leftarrow$ remove the cache entry with the least popularity in $node$;
cache $leafnode$ in $node$;
$nextNode \leftarrow$ access the next cache node at lower level for $outCacheEntry$;
if(cache space is available in $nextNode$)
    cache $outCacheEntry$ in $nextNode$;
    return;
elseif($nextNode$ is at the lowest cache level AND popularity of $leafnode >$the least popularity in $nextNode$)
    remove the cache entry with least popularity in $nextNode$;
    cache $outCacheEntry$ in $nextNode$;
    return;
else
    InsertCache(nextNode,outCacheEntry);
End **InsertCache**

	CCB+Tree(1)	CCB+Tree(2)
p1f10	904	1388
p1f50	814	1360
p1f100	863	1479
p1f500	1105	1909
p1f1000	1461	1779
p10f10	815	1453
p10f20	855	1682
p10f50	1103	1796
p10f100	1387	1840
p50f10	973	1732
p50f20	1157	1534
p50f50	2061	2097
p100f2	862	2180
p100f5	991	1568
p100f10	1135	1680

Table 2: No of cache hits in CCB+-Tree(1) CCB+-Tree(2)

As we can see in Fig.1,Fig.2,Fig.3,Fig.4 that as the frequency of the popular points increases the average time of point query decreases. This can be justified by the observations in Table.2, with increase in the frequency of popular points the number of cache-hits increase, and hence, it results in faster search operation.

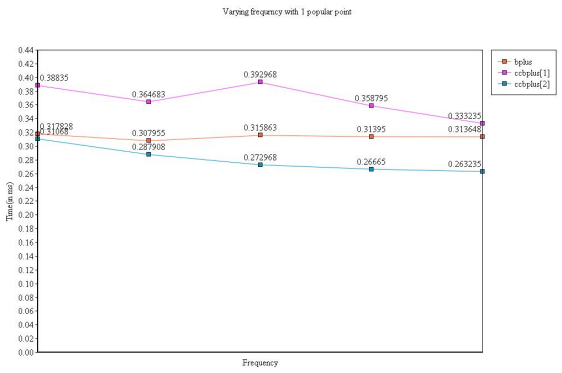

Figure 1: Varying frequency with 1 popular point

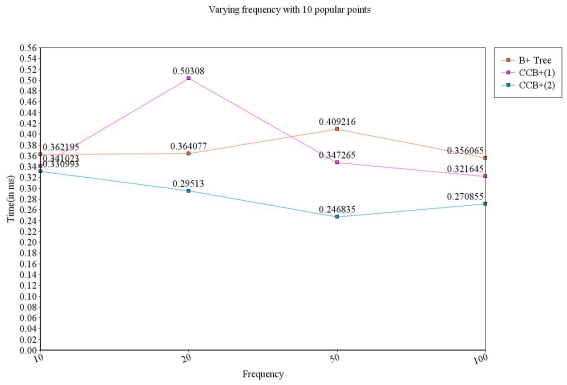

Figure 2: Varying frequency with 10 popular point

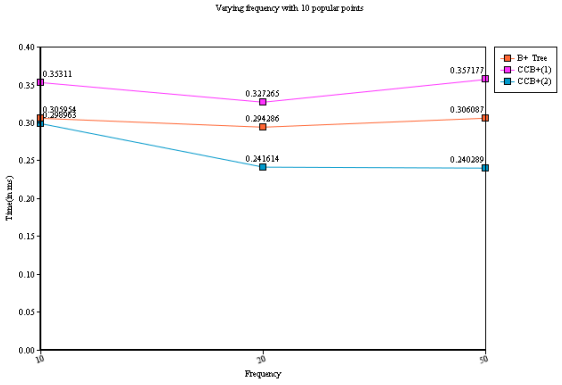

Figure 3: Varying frequency with 50 popular point

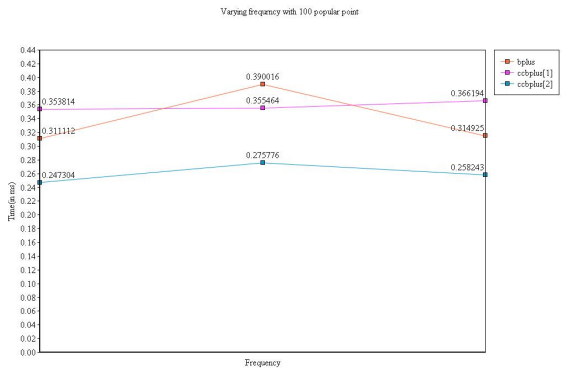

Figure 4: Varying frequency with 100 popular point

4 Conclusions

CCB+ tree is a variant of B+ tree which tries to reduce query time significantly by caching popular points(leaf nodes) closer to the root. This is helpful for huge data and few popular query points since there are lots of internal nodes which need not be traversed in order to reach the popular leaf nodes. Experiments show for the data we used(uniformly distributed points in single dimension in range [0,1]) that the CCB+ tree does not perform much better than B+-tree if the frequencies of popular points is less. But as the frequency of points increase, CCB+-tree sometimes show better results than the B+-tree. But it is not the case always, B+-tree still shows superiority over CCB+-tree(1). The reason behind this is the distribution of data we have used. Since the data is uniform, and CCB+-tree(1) also maintains the cache when we are inserting the data, which is not so helpful for our data distribution.

The overhead in search time is due to the fact that CCB+-tree has to perform extra I/Os to update the popularity counts in leaf nodes as well as the parent nodes, which does not need to be done in the regular B+-tree. Since, we have uniform data distribution and we have made queries popular, in CCB+-tree(2), we only maintain the cache in searches and the insert is same as that in B+-tree. This significantly improves the results and even shows better results than B+-tree. Thus, we conclude CCB+-tree does perform better than regular B+-tree if there are popular points in the query. But as we can see, B+-tree performs better than both instances of CCB+-tree for uniform data and uniformly distributed queries(Table-1). Looking at the results, we also can guess that the CCB+-tree(1) would have performed better if the data points were also non-uniform. Thus, we should also keep in mind that which indexing structure performs better mostly depends on the data distribution as well as how the queries are.

References

[1] Cui Yu, James Bailey, Julian Montefusco, Rui Zhang, Jiling Zhong. Enhancing the B+-tree by Dynamic Node Popularity Caching.

[2] J.Rao and K.Ross: Making B+-Trees Cache Conscious in Main Memory. SIGMOD2000.