Enhancing the B$^+$-tree by Dynamic Node Popularity Caching

Cui Yu [+] James Bailey [$] Julian Montefusco [+] Rui Zhang [$] Jiling Zhong [*]
[+] Monmouth University, USA [$] University of Melbourne, Australia [*] Troy University, USA

Keywords: databases, data structures

Abstract

We propose the Cache Coherent B$^+$-tree (CCB$^+$-tree), an indexing structure that can improve search performance compared to the traditional 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. An experimental study shows that the CCB$^+$-tree can outperform the traditional B$^+$-tree on workloads where certain queries are much more popular than the others.

1 Introduction

The B$^+$-tree is an elegant structure for indexing ordered data. It is probably the most widely implemented indexing structure in database systems, due to a variety of merits, including efficient update and search operations, scalability and robustness to various workloads. In this paper, we propose a technique to further improve the performance of the B$^+$-tree, based on the following observation. *The internal nodes of the B$^+$-tree structure are usually not fully utilized. Analysis has shown that the typical utilization rate of internal B$^+$-tree nodes is around 67% [11].*

We propose a variant of the B$^+$-tree, called the Cache Coherent B$^+$-tree (CCB$^+$-tree), which improves the search performance of the B$^+$-tree by making use of the underutilized node space. The CCB$^+$-tree uses the free space in internal nodes of the B$^+$-tree to cache pointers to popular leaf nodes. E.g. if a pointer to the most popular leaf node is cached in the root of the tree, then a query for records stored in that leaf node does not need to access any other node, which greatly reduces the I/O cost. The popularity of each leaf node is maintained after queries and the more popular node addresses are cached closer to the root. This structure is especially useful for workloads where a few values are queried much more frequently than other values and such "popular" queries may change over time. Indeed, many real applications have highly skewed data distribution such as heavy hitters in data streams [6], and data distribution in data streams usually change very quickly.

2 Tree Structure

Assuming familiarity with the traditional B^+-tree, we begin by describing the structure of the CCB^+-tree.

2.1 Node Structure

The structure of a CCB^+-tree is similar to that of a B^+-tree, with the addition of two types of information, to accommodate cache entries and maintain popularity information. An internal node has an additional parameter, the number of cached entries in the node. A leaf node has an additional parameter, the popularity of the node. The popularity of a node is the number of times the node has been read. The nodes with the most reads have the highest popularity. The structures of an internal node and a leaf node are described below. For ease of reference, internal nodes containing caching information will sometimes be referred as *cache nodes*; and the tree levels used for caching will sometimes be referred as *cache levels*. A CCB^+-tree of order M is formally defined as follows:

An internal node consists of two lists L1 and L2, where L1 is a list of pointers and keys: $(p_0,K_1,p_1,K_2,p_2,...,K_s,p_s)$; this part is the same as in a standard B^+-tree of order M, where $M = 2a - 1$ for some integer $a > 1$ 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 integer $i = 1, 2, ..., r$ and $r \leq (M\text{-}s)/2$; p_i is a pointer to a leaf node; K_i and K'_i are the smallest and largest keys in the leaf node pointed by p_i; c_i is a popularity count of the leaf node pointed by p_i. These elements are cache entries. List L2 can contain at most (M-s)/2 elements. This means that if L2 is non-empty and has (M-s)/2 cache entries, when a new (pointer, router) pair is added to L1, one element of L2 needs to be removed to create space for the insertion to fit in L1. If L2 is empty, the node will be split exactly in the same way as for standard B^+-trees.

Leaf nodes have the form $(C_t, \langle K_i, p_i\rangle, p_x)$, where C_t is a popularity count; p_x is a pointer to the next leaf node; $\langle K_i, p_i\rangle$ are the standard leaf node entries where $i = 1, 2, ..., N$, $N \leq M$; K_i is a key value; p_i is a pointer to a record having key K_i.

2.2 Cache Entry

A cache uses two vacant adjacent entry slots, which consist of two key slots and two child-pointer slots, in an internal node. The two key slots store the key range of the cached node, while the two child pointer slots store the popularity and address of the cached node, respectively. Cache entries are stored backwards, starting from the end of the page, allowing efficient addition and deletion. Figure 1 illustrates the arrangement, showing a cache entry pointing to a leaf node whose keys are in the range, from 1 to 9. Internal nodes one level above the leaf nodes should not be used to store cache entries, since this will never save any node accesses. Please note that such cache entries maintain the locality of the cached nodes, i.e., our cache maintenance mechanisms guarantee that a cache node only contains cache entries for leaf nodes within its own sub-tree. This makes the cache entries more relevant to the searches. Therefore, our approach is different from simply setting some space aside as buffer for caching, which has no correlation with which sub-tree the cached leaf node is in.

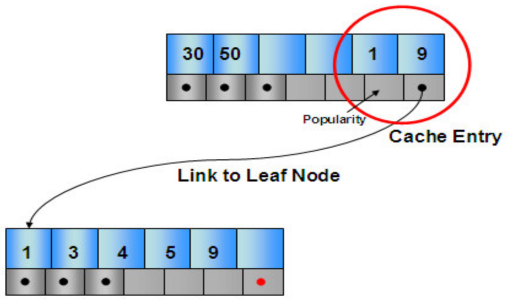

Figure 1: Cache Entry within an Internal Node

3 Basic Operations

3.1 Search

Searching a CCB$^+$-tree is similar to a search through a standard B$^+$-tree (referred to as the *Regular Search* hereafter). There are two main differences between *Regular Search* and *Cache Search*: (1) *cache Search* examines the cache entries of each encountered internal node, before continuing with the regular search; (2) The traversal path of each search is dynamically recorded in order to maintain cache access statistics. When searching for a key in an internal node, cache entries of the node will be checked first before continuing with the regular search. If the key is found within a key range of a cached leaf node, the pointer stored in the cached entry will act as a shortcut to the leaf node holding the searched key, thus reducing the search time. When the leaf node is accessed, its popularity will be increased. Each time the popularity changes for a leaf node, its value is compared with the cached popularity values in the node higher up the B$^+$-tree. If the popularity of the leaf node has become greater than the popularity of the least popular cache entry in the parent node, then the cache entries need to be updated to address this change. This is accomplished by switching the updated cache entry with the least popular cache entry of the parent node and recursively continuing up the tree, until the condition is met that the entries in the parent node are all greater than the entries in the child node. In the scenario that the search key is not within any cached key range, the search traverses all the levels of the tree, as is the case for a traditional B^+-tree.

3.2 Adding New Cache Entries

When caching a leaf node, we make use of the traversal path of the search that has just accessed the leaf node. This path allows us to place a cache entry in the correct path from the root to the lower level internal nodes. The algorithm for this, **AddNewCache(***leafnode, path***)**, is shown below.

```
Algorithm 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)
6          cache leafnode in currentNode;
7          return;
8      elseif (currentNode is at the lowest cache level AND
           popularity of leafnode > the least popularity in currentNode )
9          remove the cache entry with least popularity;
10         cache leafnode in currentNode;
11         return;
12     elseif (popularity of leafnode > the least popularity in currentNode )
13         InsertCache(currentNode, leafnodeInfo);
14         return;
15     else
16         path ← next of path;
17         currentNode ← node with address path.offset;
 End AddNewCache
```

The input *leafnode* contains the information of a leaf node to be cached. The input *path* stores the path of a search to *leafnode* starting from the root node. We consider the nodes in the path one after another for inserting the cache entry. For a node being considered (*currentNode*), if there is available space for a cache entry, then *leafnode*'s cache entry is inserted to *currentNode*. Otherwise, if *leafnode*'s popularity is higher than the cache entry with the least popularity in *currentNode*, then the function **InsertCache( )** is invoked, which inserts *leafnode*'s cache entry to *currentNode* and inserts the cache entry with the least popularity to lower level nodes down the *path*. An exception is that when *currentNode* is at level 2 (leaf nodes are at level 0), the cache entry with the least popularity will simply be discarded rather than being inserted to lower level nodes, since cache entries in level 1 nodes do not have any benefit. Details of the function **InsertCache**( ) is discussed in the next subsection.

3.3 Insert

Inserting of a new entry to a leaf node needs to be handled carefully, especially when splitting occurs. One question is: if a leaf node N1 is split and a new node N2 is created from the split, should the popularity of

N1 be split as well, or should it remain the same as the initial popularity of the splitting node? Because we do not know which keys actually contribute to the popularity of a leaf node, we adopt a safe strategy of assigning popularity, which gives N1's old popularity value to both N1 and N2 after the split. Although this may initially result in an overestimate of popularities of the nodes, the popularities will adjust over time, as new accesses occur and as the caching information gets updated.

The process of splitting a leaf node N1 into N1 and N2 involves three steps: (i) distributing the leaf node entries evenly between N1 and N2 and setting the popularities of N1 and N2 as described above; (ii) posting an index entry for N2 to the parent node of N1; (iii) if N1 is cached, updating the key range of N1's cache entry and inserting a cache entry for N2. For step (ii), if the parent node of N1 is full, it will also split and so that another new index entry will be inserted at the higher level. Such splits may further propagate up the tree. When an index entry insertion to a cache node does not require node splitting, its cache entry with the least popularity will be removed to make space for the new index entry if there is not enough space – L2 is nonempty with (M-s)/2 cache entries (ref to Section 2.1). If a removed cache entry is not from the lowest cache level, it will be inserted to the next lower level of the tree, using the **InsertCache( )** algorithm; otherwise, it will be discarded. When an index entry insertion at a cache level causes node splitting, the node must be full of index entries – L1 has M index entries (ref to Section 2.1), leaving no space for cache entries – L2 is empty. Then, only the index entries will be redistributed during node splitting. For step (iii), we use the **AddNewCache( )** algorithm described in Section 3.2 to add a cache entry for N2. Since N2 has the same popularity as the cache entry of N1, finding the node to cache N2 basically means finding the node containing the cache entry of N1. Therefore, the key range of N1's cache entry can be updated with almost no additional cost.

When inserting N2's cache entry into the tree, we examine the search path that leads to N2 as explained in the **AddNewCache( )** algorithm. Starting from the root node in the search path, if there is cache entry space available, N2's cache entry is inserted there; otherwise, the cache entry with the least popularity is moved to the lower cache level of the tree, creating space to accommodate the new cache entry – The algorithm is **InsertCache(***node*, *leafnode***)**, which describes how the cache entry with the least popularity is recursively pushed down the tree. **InsertCache(***node*, *leafnode***)** is called when the node *leafnode* is to be cached in an internal *node* but *node* has no space available for an additional cache entry.

```
Algorithm InsertCache(node, leafnode)
1  outCacheEntry ← remove the cache entry with the least popularity in node;
2  cache leafnode in node;
3  nextNode ← access the next cache node at lower level for outCacheEntry;
4  if (cache space is available in nextNode)
6       cache outCacheEntry in nextNode;
7       return;
8  elseif (nextNode is at the lowest cache level AND
       popularity of leafnode > the least popularity in nextNode )
9       remove the cache entry with least popularity in nextNode;
10      cache outCacheEntry in nextNode;
11      return;
12 else
13      InsertCache(nextNode, outCacheEntry);
End InsertCache
```

If the cache node splits, the cache entries must be split as well and there will be a lot of space available for caching. To avoid the cost of moving cache entries between different levels, the unused cache space is kept vacant for (future) new cache entry insertions. This will temporally allow regional cache entries not to strictly follow the rule of "the more popular, the closer to root", until other cache insertions push them down when they are less popular. The basic algorithms for node split are similar to those of the traditional B^+-tree, but can also handle updating the popularity of a splitting leaf node and the ability to split the cache entries for a splitting cache node. They are straightforward and therefore omitted here.

3.4 Delete

Deletion of a key also needs more care if node merging is required. When two leaf nodes are combined, the new leaf node assumes the larger of the two popularities, which helps limit cache redistribution. When two leaf nodes merge, if neither of them is currently cached, there is nothing to do about caching. If one leaf is currently cached, it must be the one with higher popularity, so we only update its cache information to reflect the merged leaf node. If both leaf nodes are currently cached, the cache entry with higher popularity will be kept in cache node, so we update its information to reflect the merged leaf node and remove the cache entry for the less popular leaf node since it no longer exists. To save cache redistribution cost, the new 'vacant' cache space will be kept available for a later cache insertion. This temporally allows cache entries break the rule of "the more popular, the closer to root", until other cache insertions pushing it down when it is less popular. Further, it is indeed not necessary to update the key range in the cache entry especially if not both old leaf nodes were cached. The change can be delayed till modification is unavoidable because of other reasons, such as updating popularity or address. The algorithms for node merging are similar to those of traditional B^+-tree, so we omit them here.

4 Experimental Study

We next report the results of various tests to evaluate the improvements of using cache entries in the CCB^+-tree searches. The *Cache Searches* of the CCB^+-tree were compared to the *Regular Searches* of the traditional B^+-tree. Both types of searches were run on trees containing exactly the same record information, based on the same test data input. The result reported for each experiment is the average result of running 1000 queries, with varying percentages of popularity, buffer size used to store tree node information in main memory, and data size used to build the individual B^+-trees. The experiments were run on a machine using Linux 2.6.13-15 with a 1.30 GHz Intel(R) Celeron(R) processor.

4.1 I/O Performance

We first compare the I/O costs of *Cache Search* and *Regular Search*. Trees were built with page sizes of 1KB, 2KB, and 4KB, using either 1000K or 2000K data records stored in the tree. For the 1KB/2KB/4KB page size, the internal node fanout is about 90/180/360 and each leaf node contains about 60/120/240 records, respectively. In the 1000 queries, 50% were considered popular, in that the leaf nodes they accessed were cached in the CCB^+-tree. Each search was run with different buffer sizes of 10 to 100 in increments of 10. The results for experiments with 1KB and 4KB page sizes are shown in Figures 2 and 3.

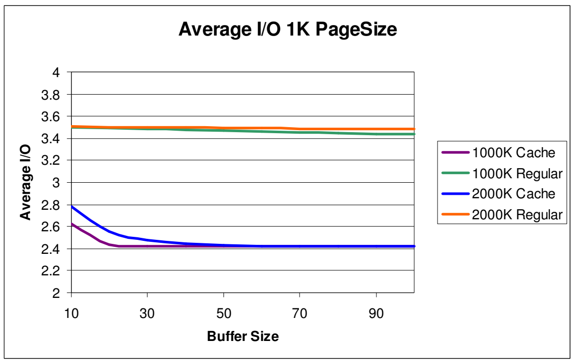

Figure 2: Average I/O for 1K PAGESIZE

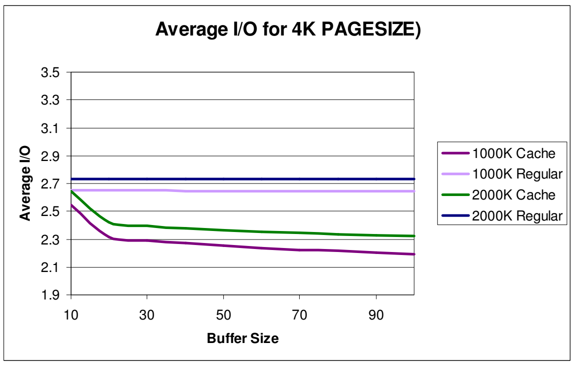

Figure 3: Average I/O for 4K PAGESIZE

In all the settings, *Cache Search* has considerable fewer I/Os than *Regular Search*, and their numbers of I/Os both drop as the buffer size increases. The result on the experiment with 2KB page size has a similar behaviour and we omit the figure.

4.2 Execution Time Performance

Implementing *Cache Search* incurs overhead including the actual search through cache entries, updating of popularity, and possible cache insertion or updating. We next analysed 20,000 queries to test the impact of this overhead. Figure 4 shows the total response time (in seconds) of 20,000 queries using *Cache Search* and *Regular Search* as we vary the buffer size and set the page size as 1KB. We also plotted the time of *Cache Search* without counting the above overhead for comparison. We observe that *Cache Search* has a good margin of improvement over *Regular Search* in most cases despite of the overhead. The speedup is up to 47% in this experiment. The overhead is negligible compared to the total response time. This shows that the benefit we gain through the cached entries overweighs the overhead they incur. We have also performed these experiments on other page sizes such as 2KB and 4KB. The results show very similar behaviour and we omit the figures due to space limit.

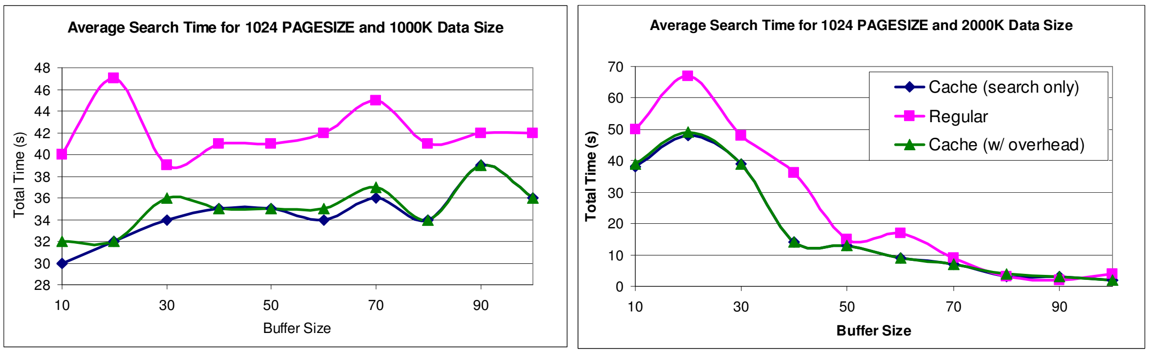

Figure 4: Query Processing Time

4.3 Effect of Query Popularity

This experiment studies the effect of query popularity. We vary the percentage of popular queries, i.e., queries on entries cached in the CCB$^+$-tree and report the I/Os in Figure 5. The number of I/Os decreases as

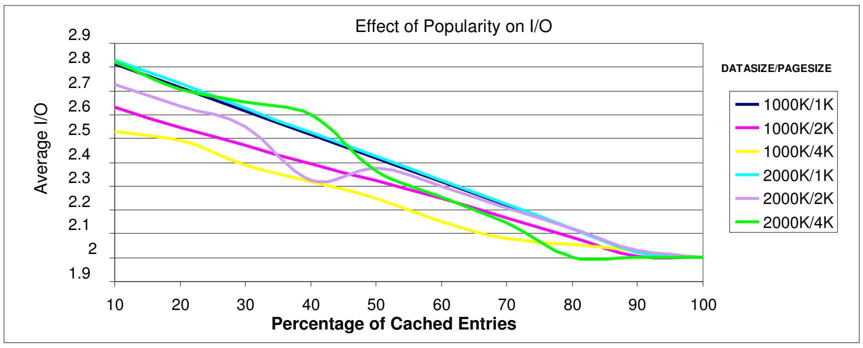

Figure 5 : Effect of Popularity on Average I/O for Cache Search

more queries are on popular entries. This is because the more the queries on cached entries, the more I/Os can be saved in the search. When popular queries approach high percentage, there is significant I/O cost save (up to 30%). As we have discussed in the introduction, data in many applications (such as data streams) show a highly skewed distribution and therefore can benefit from the CCB$^+$-tree. Furthermore, this experiment again verifies that the benefit of cache entries overweighs the overhead introduced by them.

5 Related Work

Early work in [5, 8] notes that nodes in high levels of a B$^+$-tree (particularly the root) are underutilized and so allowing a variable order for nodes at different depths is suggested. Work in [7,8] has analysed the node utilization of B$^+$-trees. Since the typical utilization rate is under 70%, there is still potential to improve the performance of B$^+$-trees. The B*-tree [9] is a variation of the B$^+$-tree, which improves the node utilization by redistributing entries between sibling nodes before insertion or deletion of nodes. In the context of maintaining multiple versions of the data in a B$^+$-tree, it has been proposed in [3] to use, where possible, the free space in nodes. Other studies [4, 12] have focused on adapting the B$^+$-tree structure to obtain better cache behaviour. Work in [1] presented buffer-trees, which like the CCB$^+$-tree, use the concept of attaching extra buffer space to nodes in a tree structure. In buffer trees, this space is intended to facilitate ``batching'' of pending insertions/deletions and thus improve I/O performance, whereas buffer space in a CCB$^+$-tree is tied to the concept of popularity and is used for facilitating access to popular leaf nodes. Our idea is also loosely related to ``access aware'' data structures, such as the ``move-to-the-front'' method for updating lists [10], whereby any item retrieved from a list is placed at its head.

6 Conclusions and Discussions

We proposed a variation of the B$^+$-tree, the Cache Coherent B$^+$-tree (CCB$^+$-tree), which utilizes the unused space in the internal nodes of the B$^+$-tree to store cache entries. The cache entries serve as shortcuts to locate the leaf nodes and therefore can save search cost for queries on popular leaf nodes. This structure is especially suitable for workloads where certain queries are much more popular than the others. We have presented how to perform insertion, deletion, searching and maintenance of the cache entries in the CCB$^+$-tree. We have also experimentally verified that the CCB$^+$-tree outperforms the B$^+$-tree in our targeted workloads. The target workloads of the CCB$^+$-tree are those where there is only a single user or where strict concurrency control is unnecessary, e.g., in data stream applications [13] or information retrieval tasks. Implementing the CCB$^+$-tree in a full-fledged DBMS requires concurrency control schemes.

Additional locks may need to be obtained in addition to those needed for the traditional B^+-tree. For the search operation, if the popularity count of a leaf node needs to be incremented, a write lock on the node that contains the cache entry needs to be obtained. For the insertion operation, we need to obtain locks on the (internal) nodes on the search path so that if a split happens, we can update the key range of the cache entry for the split node. We will defer a full investigation of concurrency control and recovery mechanisms of the CCB^+-tree to future work. It is also interesting to further explore how the caching idea can be extended to other types of keys such as strings or multi-attribute keys.

7 References

[1] L. Arge. The Buffer Tree: A Technique for Designing Batched External Data Structures. *Algorithmica. 37(1):1-24, 2003.*

[2] R. Bayer and E. McCreight, Organization and Maintenance of Large Ordered Indices. *Acta Informatica. 1(3): 173-189, 1972.*

[3] B. Becker, S. Gsschwind, T. Ohler, B. Seeger and P. Widmayer. An Asymptotically Optimal Multiversion B-Tree. *VLDB Journal. 5(4): 264-275, 1996.*

[4] S. Chen, P. Gibbons, T. Mowry and G. Valentin: Fractal Prefetching B+-Trees: Optimizing Both Cache and Disk Performance. *SIGMOD 2002.*

[5] D. Comer, The Ubiquitous B-Tree. *ACM Computing Surveys. 11(2): 121-137, 1979.*

[6] G. Cormode, F. Korn, S. Muthukrishnan and D. Srivastava: Finding Hierarchical Heavy Hitters in Data Streams. *VLDB 2003.*

[7] D. Lomet, B-tree Page Size When Caching is Considered. *SIGMOD Record. 27 (3): 28-32, 1998.*

[8] T. Johnson and D. Shasha, Utilization of B-trees with Inserts, Deletes, and Modifies. *PODS 1989.*

[9] D. E. Knuth. The Art of Computer Programming. Vol. 3. Addison Wesley, 2002.

[10] J. McCabe. On Serial Files with Relocatable Records. *Operations Research. Vol. 13: 609-618, 1965.*

[11] R. Ramamkrishnan and J. Gehrke, Database Management Systems. 3rd edition. *McGraw-Hill, 2007.*

[12] J. Rao and K. Ross: Making B+-Trees Cache Conscious in Main Memory. *SIGMOD 2000.*

[13] Nick Koudas, Beng Chin Ooi, Kian-Lee Tan, Rui Zhang: Approximate NN queries on Streams with Guaranteed Error/performance Bounds. *VLDB 2004.*