Main Memory Indexing: The Case for BD-tree

Bin Cui *Beng Chin Ooi* *Jianwen Su* *Kian-Lee Tan*

[1]Department of Computer Science
National University of Singapore
3 Science Drive 2, Singapore 117543
{cuibin,ooibc,tankl}@comp.nus.edu.sg

[2]Department of Computer Science
University of California
Santa Barbara, CA 93106-5110
su@cs.ucsb.edu

Abstract

In this paper, we adapt and optimize the BD-tree for main memory data processing. We compare the memory-based BD-tree against the B^+-tree and CSB^+-tree. We present cost models for exact match query for these indexes, including L2 cache and translation lookahead buffer (TLB) miss model and execution time model. We also implemented these structures and conducted experimental study. Our analytical and experimental results show that a well tuned BD-tree is superior in most cases.

Index Terms: Main memory databases, BD-tree, B^+-tree, CSB^+-tree.

1 Introduction

The impact of advances in hardware and main memory capacity is beginning to be felt by the software community [3, 4, 6, 11].Not only is it possible to store the entirety of a database in memory now, many algorithms that were designed in the past need to be reexamined. For example, the memory-based index structure T-tree [9] is no longer attractive because of its low fanout, poor cache behavior and excessive utilization of pointers. In this paper, we reexamine the indexing issue in main memory databases.

To improve performance, an indexing structure must facilitate effective use of the L2 caches and CPU. One promising structure is the CSB^+-tree [11] that reduces the cache misses by storing all the child nodes of any given node contiguously, and keeping only the address of the first child in each node, thereby increasing the utilization of the cache line. However, while the CSB^+-tree is efficient for range queries, it does not perform well for exact match queries (compared to hash-based schemes). This is because multiple nodes need to be accessed during

the tree traversal. This prompted us to search for a structure that can combine the benefits of hashing and tree indexing.

In this paper, we present our solution to facilitate efficient query processing in main memory environment. We adapted and optimized the Bounded Disorder (BD) method introduced by Litwin and Lomet [10] for main memory processing. The BD-tree is essentially a B^+-tree where the leaf nodes are large-sized partitions, and each leaf node is organized by hash tables. We evaluated the optimized memory-based BD-tree against the CSB^+-tree and B^+-tree analytically and empirically. Our results show that a well-tuned BD-tree outperforms the other two indexes in most cases.

2 The Memory-based BD-tree

Figure 1 shows the structure of the BD-tree [10], which comprises two tiers. In the first tier, a hierarchical multi-way tree structure (e.g., B^+-tree) *range partitions* the data. The second tier begins at the leaf level of the tree, which organizes the data in each leaf node using a hashing method. Records within a leaf are not ordered.

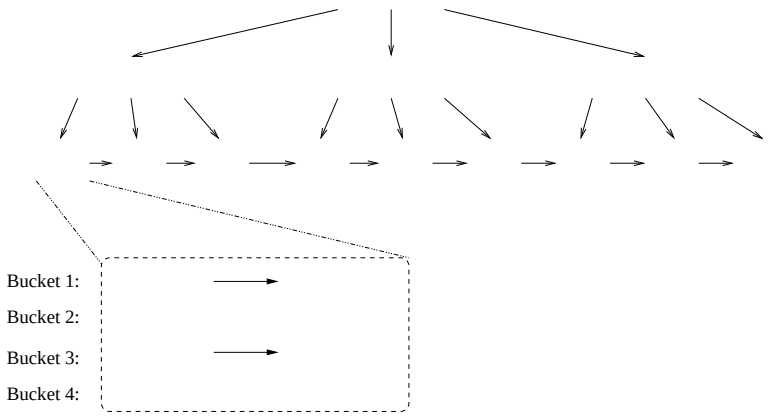


Figure 1: The BD-tree

The BD-tree is efficient for memory-based query processing. First, the leaf node size of a BD-tree is much larger than that of a B^+-tree. This means fewer internal nodes are needed, i.e. a shorter tree. Therefore fewer cache and TLB[1] misses are needed during tree traversal.

[1]The TLB is an essential part of modern processors, which keeps the mapping from logical memory address to physical memory address.

Moreover, since leaf nodes are organized as hash buckets, only the necessary buckets need to be searched. This clearly benefits exact match queries. Second, the tree structure essentially acts as a dynamic range partitioning mechanism, allowing each leaf node to have different range size. As such, only the necessary leaves need to be examined for range queries.

To optimize the BD-tree for main memory processing, we adapt the hash function with low computation cost to reduce the CPU overhead of searching the leaf nodes. Since the use of tree structure essentially acts as a dynamic range partitioning mechanism and the difference between the numbers of keys in the leaf nodes is bounded, the average performance can be accurately predicted under the uniformity assumption. Moreover, since data may be skewed or the hash functions may not distribute the data uniformly within each leaf, we allow each bucket to have at most k chains (for some integer $k >= 1$). In other words, a leaf will be split only when a new record is to be inserted into a bucket whose k chained buckets are already full. There are exceptions where the duplicated keys (hash keys) overflow the chained buckets and no splitting will help. In these cases, we allow more than k chains. For simplicity, we ignore such special cases in our discussion. Since the fanout of a main memory tree structure is typically small compared to a disk-based index, we can reduce the height of BD-tree significantly by setting a relatively large leaf size. Within the leaf, we set the bucket size to correspond to the cache line size. Note that although we can keep the leaf nodes of a B^+-tree/CSB^+-tree larger than its internal nodes, a larger leaf node size does not result in better search performance. Because a large leaf node size also means more cache misses when the node is accessed, some non-contiguous segments of the leaf need to be loaded into cache even if we use binary search. For exact match queries in the BD-tree, we only need one cache miss to access each chain bucket and at most k buckets to be retrieved in leaf level. For range queries, because the leaf partitions are ordered, we just scan the leaves that overlap the desired query ranges. To speed up the range query, we store the chained buckets contiguous with the primary buckets of the leaf in the memory, and hence reduce the TLB misses of scanning.

3 Cost Analysis

To have an insight into the performance of the B^+-tree, CSB^+-tree and BD-tree, we present analytical models to evaluate the cost of exact match query, i.e. a cache and TLB misses model and an execution time model. We model the cost of exact match query as a function of three variables: number of L2 cache misses, TLB misses and instructions. There are other factors that affect performance, such as branch mispredictions, instruction cache misses and L1 data cache misses. However these factors do not play a significant role in determining the overall cost [3]. The representative parameters used for the analysis are shown in Table 1. Some of the values were produced by measuring the performance of indexes in the experiment. And to simplify the analysis, we assume that the buckets in the leaf nodes of the BD-tree have no overflow chains.

Variable	Value	Description
c	10M	the cardinality of the dataset
$\|k\|$	4 bytes	the size of a key
$\|p\|$	4 bytes	the size of pointer
$\|r\|$	8 bytes	the size of a $< key, RID >$ pair
$\|l\|$	64 bytes	the size of a cache line (UltraSPARC processor)
$\|n\|$	32-256 bytes	the size of a node
$\|b\|$	64 bytes	the size of a hash bucket
u	70%	the average utilization of a node/bucket
d	64	the directory size of leaf partition (the BD-tree)
m_c	150 cycles	the cost of a L2 cache miss
m_t	100 cycles	the cost of a TLB miss
$\|m\|$	12 bytes	the size of metadata in the node of tree
I_b	40	number of instructions to evaluate a key and calculate next evaluation position in binary search

Table 1: Parameters for Analysis

3.1 Cache and TLB Miss Model

The L2 cache misses and TLB misses play an important role in the tree operations. For each node access, there may be several cache misses and one TLB miss for various node sizes (< virtual memory page size). The node size and tree height are two main parameters in this cost model. A larger node size implies a shorter tree, and vice versa.

To determine the height of the tree, we first present the formula for fanout, f.

$$f = \begin{cases} \lfloor u * \frac{|n|-|m|}{|k|} + 1 \rfloor & : \quad CSB^+\text{-}tree \\ \lfloor u * \frac{|n|-|m|}{|k|+|p|} + 1 \rfloor & : \quad B^+\text{-}tree/BD\text{-}tree \end{cases}$$

Also with the given size of dataset, the number of leaf node, L is :

$$L = \begin{cases} \lceil \frac{c}{u * \frac{|n|-|m|}{|r|}} \rceil & : \quad B^+\text{-}tree/CSB^+\text{-}tree \\ \lceil \frac{c}{u * d * \frac{|b|}{|r|}} \rceil & : \quad BD\text{-}tree \end{cases}$$

Therefore, the height of the tree, h, is given as follows:

$$h = \lceil \log_f L + 1 \rceil.$$

For an exact match query, we must traverse the tree from the root to the leaf to get the answer. At each level, a child node is determined by a binary search in the parent node, so the number of L2 cache lines accessed of a node is $\lceil (\log_2 (u * n_c) + 0.5) \rceil$, where n_c is the number of cache lines spanned by one node ($n_c = \lceil \frac{|n|}{|l|} \rceil$). Because we always need to access the first cache block of the node, we add 0.5 cache miss for each node access.

On the first query of the index, the number of cache misses is the cache lines accessed. However, there is a higher probability of finding some nodes in the cache for subsequent queries, e.g. the nodes in the upper levels of the tree are accessed frequently and may always reside in the cache. Re-accessing of these nodes will result in fewer cache misses. To model cache misses more efficiently, we use the Cardenas's formula [5]:

$$X(n, q) = n * (1 - (1 - 1/n)^q),$$

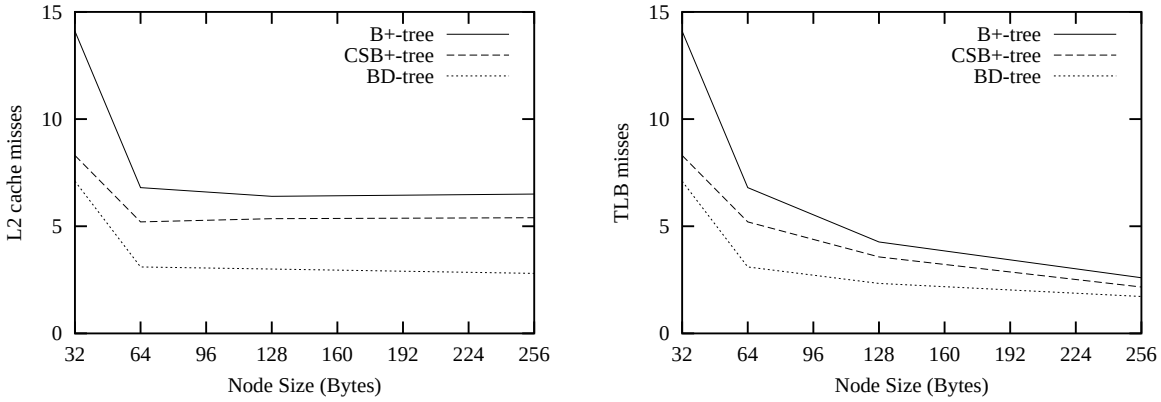
where X is the number of unique nodes accessed, n is the number of nodes available and q is the number of queries. Using this formula, the average number of uniquely accessed nodes for a query, i.e. $\frac{X(n,q)}{q}$, reduces when we increase the number of queries. We combine the Cardenas's formula with the tree structure to determine the number of unique nodes that are accessed when we traverse the tree. The number of cache misses in the leaf node of BD-tree is 1 as we assume no chained buckets in the model. Thus we can model the number of cache misses per query as:

$$C_m = \begin{cases} \frac{\sum_{i=1}^{h} X(n(i),q) * \lceil (\log_2 (u * n_c) + 0.5) \rceil}{q} & : \quad B^+\text{-}tree/CSB^+\text{-}tree \\ \frac{\sum_{i=1}^{h-1} X(n(i),q) * \lceil (\log_2 (u * n_c) + 0.5) \rceil + X(n(h),q)}{q} & : \quad BD\text{-}tree \end{cases}$$

where $n(i)$ is the number of nodes at level i in the tree ($n(i) = f^{i-1}$). The model gives the compulsory cache misses, but not the capacity cache misses [8]. Because the cache size is finite, eventually some highly accessed cache lines can be replaced after running many queries. However, with large cache size, these effects are not significant for a first-order approximation.

For an individual query, the number of TLB misses can be equal to the height of the tree, while the TLB misses may be satisfied from the L2 cache hits and top levels of the tree may stay cached for subsequent queries. Therefore, the number of the TLB misses, TLB_m can be much smaller than the tree height and also smaller than C_m because of larger TLB page size. TLB_m is represented by the following equations:

$$TLB_m = \begin{cases} \frac{C_m}{\log_2(u*n_c)+0.5} & : \quad B^+\text{-}tree/CSB^+\text{-}tree \\ \frac{C_m-1}{\log_2(u*n_c)+0.5}+1 & : \quad BD\text{-}tree \end{cases}$$



(a) cache misses (b) TLB misses

Figure 2: The number of cache and TLB misses for a single query

We show the average number of TLB and cache misses for 1000 exact match queries using the above models in Figure 2. The B$^+$-tree performs worst among these trees. Compared with the CSB$^+$-tree, the B$^+$-tree has a smaller branching factor than the CSB$^+$-tree since it needs to store more child pointers. Hence, the B$^+$-tree is taller than the CSB$^+$-tree, and incurs more TLB and cache misses. Not surprisingly, the BD-tree is best because its leaf node (partition) has more keys, thus the BD-tree is shorter. When we increase the node size, both the number of cache and TLB misses reduce. The number of cache misses is much smaller than the number of cache lines accessed, because some of the frequently accessed nodes can reside in the cache.

The number of TLB misses is fewer than that of cache misses when the node size is larger than 64 bytes, because one node access only incurs one TLB miss but several cache misses in this case. For the CSB$^+$-tree, when the node size is 64 bytes (cache line size), the number of cache misses is smallest. For the B$^+$-tree and BD-tree, the optimal node size is 128 bytes. In all cases, the cache misses do not increase much for larger node sizes. Because the tree is shorter, more proportional accessed nodes can be resident in the cache, although more cache lines need to be accessed. The number of cache misses is a compromise of these factors.

3.2 Execution Time Model

The time to execute a single query (T_Q) includes computation time (T_C), L2 cache misses time (T_{CM}) and TLB misses time (T_{TLBM}). Thus, we can estimate the execution time T_Q as:

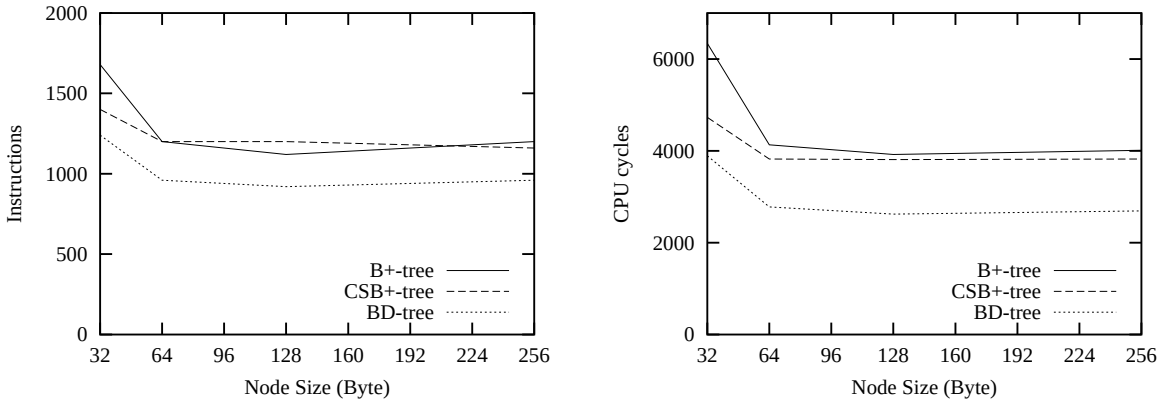
$$T_Q = T_C + T_{CM} + T_{TLBM}.$$

The above equation ignores the effects of out-of-order execution which can hide a portion of cache miss latency by continuing to execute the instructions out-of-order on the processor [3]. It is easy for us to get T_{CM} and T_{TLBM} because we know the number of TLB and cache misses from the TLB and cache miss model. We obtain the cost of each TLB and cache miss using the Calibrator Tool [1], e.g. a L2 cache miss latency m_c is 150 cycles and a TLB miss latency m_t is 100 cycles for SUNFire 4800.

The computation cost is the time taken to execute all the instructions for a query. The number of instructions executed at each level is mainly the binary search inside the node, so we can give the equation to calculate the number of total instructions I_Q as follows:

$$I_Q = \begin{cases} I_b * ((h-1) * \lceil \log_2 (u * \frac{|n|-|m|}{|k|+|p|} + 1) \rceil + \lceil \log_2 (u * \frac{|n|-|m|}{|r|} + 1) \rceil) & : \quad B^+\text{-}tree \\ I_b * ((h-1) * \lceil \log_2 (u * \frac{|n|-|m|}{|k|} + 1) \rceil + \lceil \log_2 (u * \frac{|n|-|m|}{|r|} + 1) \rceil) & : \quad CSB^+\text{-}tree \\ I_b * ((h-1) * \lceil \log_2 (u * \frac{|n|-|m|}{|k|} + 1) \rceil + \lceil \log_2 (u * \frac{|b|}{|r|} + 1) \rceil) & : \quad BD\text{-}tree \end{cases}$$

where I_b is the number of instructions required to evaluate a key and select the next position in a binary search. The Perfmon tool [2] can be used to count the number of instructions/cycles of an evaluation. Hence, we can get the number of instructions for an exact match query using the above equation. The CPU cycles per instruction (CPI) used in the model is equal to 2. The number of instructions and total CPU cycles for a query are shown in Figure 3.



(a) Instructions (b) CPU cycles

Figure 3: The number of instructions and cycles for a single query

Figure 3 (a) shows the number of instructions executed versus node size per query. The number of instructions is quite stable when the node size is larger than 64 bytes, because we execute binary search in each node and the overall computation cost is similar. More instructions are incurred when the node size is equal to 32 bytes, because the binary search is not efficient when the node size is too small. The total execution time is the combination of TLB and cache misses and CPU cost. When the node size is equal to the cache line size, the CSB$^+$-tree performs best, while 128 bytes is an optimal node size for B$^+$-tree and BD-tree. Clearly, the BD-tree performs best among these trees, followed by the CSB$^+$-tree and B$^+$-tree.

4 A Performance Study

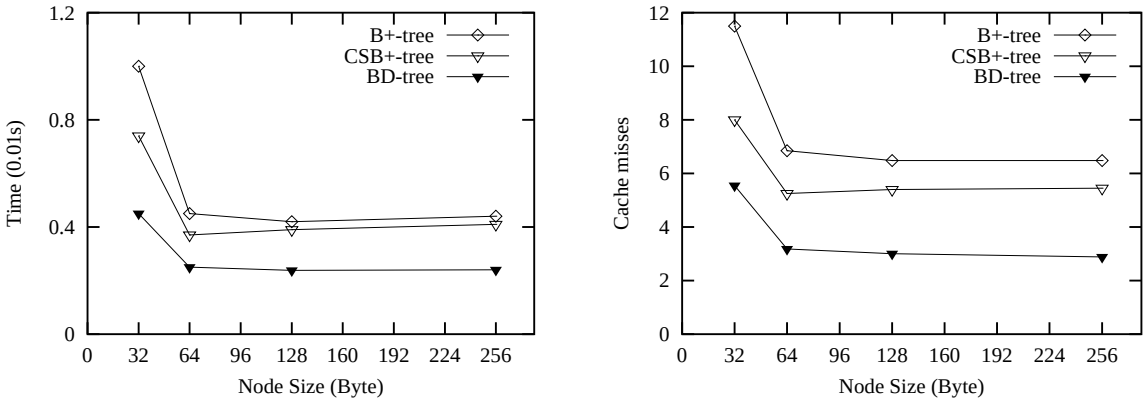
In this section, we present an experimental evaluation on SUNFire 4800, which has 750MHz CPU, 16 GB RAM and 8M L2 cache. The dataset we used consists of 10 million unique integers. The data in the structures consists of $< key, pointer >$ pairs, both 4 bytes long, and the keys are uniformly distributed. We also conducted sensitivity analysis on the techniques by varying distributions of datasets, cardinarlity of datasets, and so on. For a full set of of the experimental study, readers are referred to [7].

The BD-tree has two extra parameters, the number of buckets in leaf and the bucket size. Our preliminary study to tune these parameters shows that the exact match query performance

of BD-tree improves with a large number of buckets. On the other hand, the large leaf size may reduce the performance of range queries with small query range. We found that the performance of tree is near optimal when the leaf has 64 buckets each of size 64 bytes, and we use it as the default values in our study here. Moreover, we allow at most a chain of two buckets (i.e., $k = 2$). All the buckets in a leaf node are allocated in the contiguous memory space to minimize TLB misses.

4.1 On Exact Match Query

In this experiment, we study the effect of node size on the performance of exact match query of the various schemes by varying the node size from 32 bytes to 256 bytes. This also serves to tune the various schemes for optimal settings of node sizes. We conducted 1000 operations of exact match queries, and recorded their performance. The results are shown in Figure 4 and it is clear that the performance of the three indexes is consistent with the analytical results.



(a) Elapsed time of exact match

(b) Cache misses of exact match

Figure 4: Performance of exact match query

Figure 4 (a) shows the overall elapsed time of exact match queries. For all tree-based schemes, when the node size is too small (e.g, 32 bytes), the fanout becomes so small that it results in a very tall tree. This leads to more TLB and cache misses as the tree is traversed. When the node size increases, the number of TLB and cache misses decreases. For the B^+-tree, its optimal performance for exact match queries occurs when the node size is equal to 128 bytes. For the CSB^+-tree, we observe that it is optimal when the node size is 64 bytes which is

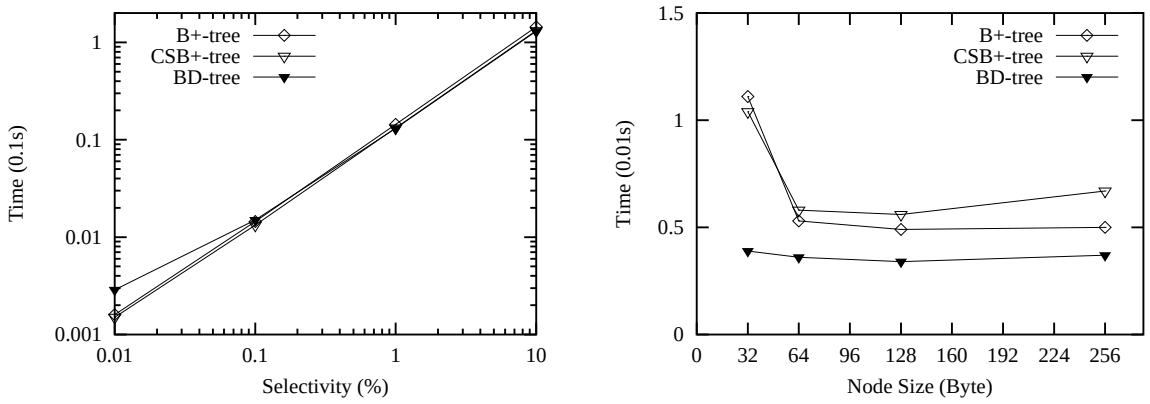
the cache line size of L2. Same as the B^+-tree, the BD-tree yields optimal performance when the internal node size is equal to 128 bytes. Comparing these various B^+-tree based structures, we see that the CSB^+-tree is better than the B^+-tree in all cases, but the difference is not significant when the node size is larger than 128 bytes. This result is consistent with the findings in the analytical study. The BD-tree performs best and is about 40% better because of the shorter tree traversal and the efficient bucket access in the leaf level.

In Figure 4 (b), we show the numbers of average cache misses of exact match query, which were obtained by Perfmon [2]. Because we did not flush the L2 cache after each query and the L2 cache can be fairly large on modern machines, running many queries on the index will eventually result in fewer cache misses per query, e.g. some highly accessed cache lines can always reside in the L2 cache. Note that there is no index node residing in the cache before the operations. So smaller index size can benefit more from this mechanism as the upper levels of the tree can always reside in the cache.

Based on the results of the previous experiment, we pick the node size for the BD-tree and B^+-tree to be 128 bytes, the node size for the CSB^+-tree to be 64 bytes. These will be used as the default settings for the subsequent experiments.

4.2 On Range Query

In this experiment, we study the various schemes' performance on range queries. We vary the selectivities of the queries ranges from 0.01% to 10% of the total number of keys. The result is shown in Figure 5 (a). The cost of range queries falls into two parts, tree traversal to locate the leaf node containing the first record and scanning all leaf nodes that contain the answers. The CSB^+-tree is always better than the B^+-tree, because the CSB^+-tree can locate the leaf faster and fewer TLB misses are incurred for scan, as the leaf nodes with the same parent node are stored sequentially in memory. Although the BD-tree can locate the leaf node much faster, the disorder within the leaf incurs some additional cost. Therefore, when the range selectivity is small, e.g. 0.01% selectivity, the BD-tree shows poorer results because it needs to access the whole leaf. On the other hand, the BD-tree performs as well as the CSB^+-tree and better than the B^+-tree when the selectivity is larger than 0.1% in our experiment. Each



(a) Elapsed time of range query

(b) Elapsed time of insertion

Figure 5: Performance of range query and insertion

leaf node of the BD-tree has much more keys than the other two trees, and hence the scan of leaf nodes causes fewest TLB misses.

4.3 On Insertion

We also study the insertion performance of the various schemes. Figure 5 (b) shows the result of 1000 insertion for various node sizes. In all cases, we observe similar performance: the BD-tree is the most efficient, the CSB$^+$-tree is the worst and the B$^+$-tree is slightly better than the CSB$^+$-tree. The CSB$^+$-tree is worse than the B$^+$-tree for the following reasons. The insertion cost has three parts: search cost, sort cost and the split cost. The split cost of the CSB$^+$-tree includes copying a complete node group, whereas a node split of the B$^+$-tree involves creating a new node. The BD-tree performs better than the B$^+$-tree because the leaf partitions of the BD-tree have much more keys than those of the CSB$^+$-tree and B$^+$-tree, thus the tree is shorter; at the same time inserting a key within a leaf is also very efficient as we only need to calculate the hash value and insert the key into the target bucket.

5 Conclusion

In this paper, we have revisited the problem of accessing data in main memory databases. We adapted and optimized the BD-tree [10] to facilitate fast search in main memory environment.

The BD-tree taps on the strengths of a hierarchical tree structure to evenly distribute and range partition the keys (facilitating efficient range queries processing) and the efficiency of hash-based methods for processing exact match queries. We studied the BD-tree against the B^+-tree and CSB^+-tree analytically and empirically. Our results showed that the BD-tree is a promising index structure for memory-based processing.

References

[1] The calibrator tool. http://www.cwi.nl/ manegold/Calibrator/, 1999.

[2] *The Perfmon Tool.* http://www.cps.msu.edu/ enbody/perfmon.html, 1999.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbms on a modern processor: Where does time go. In *Proc. 25th VLDB Conference*, pages 266–277, 1999.

[4] P. Bohannon, P. Mcllroy, and R. Rastogi. Main-memory index structures with fixed-size partial keys. In *Proc. of the ACM SIGMOD*, pages 163–174, 2001.

[5] A. Cardenas. Analysis and performance of inverted database structures. In *Communication of ACM*, volume 18, pages 253–264, May 1975.

[6] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proc. of the ACM SIGMOD*, pages 139–150, 2001.

[7] B. Cui, B. C. Ooi, J. W. Su, and K. L. Tan. *Using BD-tree for Main Memory Processing.* Technical Report, School of Computing, National University of Singapore, 2003.

[8] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan kauffman, 1998.

[9] T. Lehman and M. Carey. A study of index structures for main memory database management systems. In *Proc. 12th VLDB Conference*, pages 294–303, Kyoto,Japan, 1986.

[10] W. Litwin and D. Lomet. The bounded disorder access method. In *Proc. of the 17th ICDE*, pages 38–48, 1986.

[11] J. Rao and K. Ross. Making b+-trees cache conscious in main memory. In *Proc. of the ACM SIGMOD*, pages 475–486, 2000.