Performance Evaluation of Main-Memory R-tree Variants

Sangyong Hwang[1], Keunjoo Kwon[1], Sang K. Cha[1], Byung S. Lee[2]

[1.] Seoul National University
{syhwang, icdi, chask}@kdb.snu.ac.kr
[2.] University of Vermont
bslee@cs.uvm.edu

Abstract. There have been several techniques proposed for improving the performance of main-memory spatial indexes, but there has not been a comparative study of their performance. In this paper we compare the performance of six main-memory R-tree variants: R-tree, R*-tree, Hilbert R-tree, CR-tree, CR*-tree, and Hilbert CR-tree. CR*-trees and Hilbert CR-trees are respectively a natural extension of R*-trees and Hilbert R-trees by incorporating CR-trees' quantized relative minimum bounding rectangle (QRMBR) technique. Additionally, we apply the optimistic, latch-free index traversal (OLFIT) concurrency control mechanism for B-trees to the R-tree variants while using the GiST-link technique. We perform extensive experiments in the two categories of sequential accesses and concurrent accesses, and pick the following best trees. In sequential accesses, CR*-trees are the best for search, Hilbert R-trees for update, and Hilbert CR-trees for a mixture of them. In concurrent accesses, Hilbert CR-trees for search if data is uniformly distributed, CR*-trees for search if data is skewed, Hilbert R-trees for update, and Hilbert CR-trees for a mixture of them. We also provide detailed observations of the experimental results, and rationalize them based on the characteristics of the individual trees. As far as we know, our work is the first comprehensive performance study of main-memory R-tree variants. The results of our study provide a useful guideline in selecting the most suitable index structure in various cases.

1. Introduction

With the emergence of ubiquitous computing devices (e.g., PDAs and mobile phones) and the enabling technologies for locating such devices, the problem of managing and querying numerous spatial objects poses a new scalability challenges [JJ+01]. Since such environment involves a huge number of spatial updates and search operations, existing disk-resident DBMS may not scale up enough to meet the high performance requirement. In this regard, main-memory DBMS (MMDBMS) promises a solution to the scalability problem as the price of memory continues to drop.

MMDBMS aims at achieving high transaction performance by keeping the database in main memory and limiting the disk access only to the sequential log writing and occasional checkpointing. Recent research finds that MMDBMS accomplishes up

to two orders-of-magnitude performance improvements over disk-resident DBMS by using MMDB-specific optimization techniques. For example, the differential logging scheme improves the update and recovery performance of MMDBMS significantly by enabling fully parallel accesses to multiple logs and backup partition disks [LKC01]. Furthermore, with the primary database resident in memory without the complex mapping to disk, MMDBMS can focus on maximizing the CPU utilization. Techniques have been proposed to improve the search performance of B+-trees by utilizing the L2 cache better [RR00, CGM01, BMR01].

For multidimensional databases, our previous work on the cache-conscious R-tree (CR-tree) focuses on an inexpensive compression of minimum bounding rectangles (MBRs) to reduce L2 cache misses during a main-memory R-tree search [KCK01]. Specifically, the CR-tree uses a *quantized relative representation of MBR* (QRMBR) as the key. This compression effectively makes the R-tree wider for a given index node size, thus improving the search performance with reduced L2 cache misses. To handle dimensionality curse for the high-dimensional disk-resident R-tree, a similar technique called A-tree has been proposed independently [SY+02].

To handle concurrent index updates in real-world database applications while leveraging off-the-shelf multiprocessor systems, we have previously proposed the *optimistic latch-free index traversal* (OLFIT) as a cache-conscious index concurrency control technique that incurs minimal cache miss overhead [CH+01]. A conventional index concurrency control like lock coupling ([BS77]) pessimistically latches index nodes on every access, and incurs many coherence cache misses on shared-memory multiprocessor systems**.** OLFIT, based on a pair of node read and update primitives, completely eliminates latch operations during the index traversal. It has been empirically shown that OLFIT combined with the link technique ([LY81]) scales the search and update performance of B+-tree almost linearly on the shared-memory multiprocessor system.

To provide a useful guidance on selecting the most appropriate main memory spatial index structure in different cases, this paper investigates the search and update performance of main-memory R-tree variants experimentally in the sequential and concurrent access environments. To ensure a fair comparison, some of the existing R-tree variants need to be upgraded. For this purpose, we first apply the QRMBR technique to R*-tree and Hilbert R-tree and call the resulting trees the CR*-tree and the Hilbert CR-tree, respectively. Thus, the main-memory R-tree variants consist of R-tree, R*-tree, Hilbert R-tree, CR-tree, CR*-tree, and Hilbert CR-tree. Additionally, we apply the OLFIT to these R-tree variants, and for this we use the GiST-link technique instead of the B-link technique [KMH97].

In the sequential access experiments, the CR*-tree shows the best search performance and the Hilbert R-tree shows the best update performance. Others (i.e., R-tree, R*-tree, CR-tree) are significantly below the two. The concurrent access experiments confirm the efficacy of the OLFIT in the scalability of search and update performance. The result shows that the Hilbert CR-tree is the best in the search performance if the data is uniformly distributed whereas CR*-tree is the best if the data is skewed, and the Hilbert R-tree is the best in the update performance. In both experiments, we

judge that the Hilbert CR-tree is the best overall considering *both* the search and the update.

This paper is organized as follows. Section 2 briefly introduces the QRMBR and the OLFIT. Section 3 describes how we implement main-memory R-tree variants with the QRMBR technique. Section 4 elaborates on the concurrency control of main-memory R-trees. Section 5 presents the experimental result of index search and update performance for sequential and concurrent accesses. Section 6 summarizes the paper and outlines further work.

2. Background

2.1. Quantized relative representation of an MBR for the CR-tree

The quantized relative representation of an MBR (QRMBR) is a compressed representation of an MBR, which allows packing more entries in an R-tree node [KCK01]. This leads to a wider index tree, better utilization of cache memory, and consequently faster search performance. The QRMBR is done in two steps: representing the coordinates of an MBR relative to the coordinates of the "reference MBR" and quantizing the resulting relative coordinates with a fixed number of bits. The reference MBR of a node encloses all MBRs of its children. Relative coordinates require a smaller number of significant bits than absolute coordinates and, therefore, allow a higher compression in the quantization step.

The high performance of the QRMBR technique comes from the following two points. First, the compression is computationally simple and doable only with the data already cached, that is, the reference MBR and the MBR to be compressed. Second, the overlap-check between a QRMBR and a query rectangle can be done by computing the QRMBR of the query rectangle and comparing it with the given QRMBR. This property allows the overlap-check to be done by compressing the query rectangle once instead of decompressing the QRMBR of every node encountered during the search.

2.2. OLFIT concurrency control of main-memory B+-trees

Concurrency control of main-memory indexes typically uses latches placed inside an index node. A latch operation involves a memory-write, whether the operation is for acquiring or releasing a latch and whether the latch is in a shared-mode or an exclusive-mode. In the case of a conventional index concurrency control, a cache block containing a latch is invalidated even if the index is not updated. The optimistic, latch-free index traversal (OLFIT) concurrency control reduces this kind of cache misses by using two primitives for node accesses: `UpdateNode` and `ReadNode` [CH+01]. These primitives use a version as well as a latch in each node as shown in the following algorithms.

Algorithm `UpdateNode`
U1. Acquire the `latch`.
U2. Update the content of the node.
U3. Increment the `version`.
U4. Release the `latch`.

Algorithm `ReadNode`
`R1`. Copy the value of the `version` into a register.
`R2`. Read the content of the node.
`R3`. If the `latch` is locked, go to Step `R1`.
`R4`. If the current value of the `version` is different from the copied value in the register, go to Step `R1`.

Step R3 and Step R4 of `ReadNode` guarantee that transactions read a consistent version of a node without holding any latch. Specifically, Step R3 checks if the node is being updated by another transaction, and Step R4 checks if the node has been updated by another transaction while the current transaction is reading the content in Step R2. Consequently, if the read operation in Step R2 is interfered by any other concurrent update, the transaction cannot pass either Step R3 or Step R4 since the condition of either one becomes true.

Provided with the two primitive operations, Cha et al. combines the B-link technique with the primitives to support the concurrency control of B+-trees. The B-link technique places a high key and a link pointer in each node. A high key is the upper bound of all key values in a node, and a link pointer is a pointer to the right neighbor of the node [LY81]. The purpose of a link pointer is to provide an additional method for reaching a node, and the purpose of a high key is to determine whether to traverse through the link pointer or not. All splits are done from left to right, and a new node splitting from a node becomes its right neighbor. These link pointers make all nodes that split from a node reachable from the original node and make the correct child node reachable without lock coupling in the case of concurrent splits of nodes.

3. Main-Memory R-tree Variants

3.1. Overview

The **R-tree** is a height-balanced tree for indexing multi-dimensional keys [Gut84]. Other variants considered in this section are founded on this structure. Each node is associated with an MBR that encompasses the MBRs of all descendents of the node. The search operation traverses the tree to find all leaf nodes of which the MBRs overlap the query rectangle. On insertion of a new entry, the R-tree finds the leaf node that needs the least area enlargement of its MBR in order to contain the MBR of the new node.

The **R*-tree** is a variant of the R-tree that uses a different insertion policy and overflow treatment policy for better search performance [BK+90]. While traversing the tree for inserting a new entry, it chooses the internal node that needs the least *area* enlargement of its MBR and the leaf node that needs the least *overlap* enlargement of its MBR. However, this policy degrades the update performance because the CPU cost of finding such a leaf node is quadratic with the number of entries [BK+90]. Therefore, using the least overlap enlargement is left as an optional policy. If a node overflows then, before splitting it, the R*-tree first tries to reinsert part of the entries that are the farthest from the center of the node's MBR. This reinsertion improves the search performance by dynamically reorganizing the tree structure. However, it makes the concurrency control difficult without latching the whole tree. Compared with the split algorithm of the R-tree that considers only the area, that of the R*-tree considers the area, the margin, and the overlap, and achieves better clustering.

The **Hilbert R-tree** uses the Hilbert curve to impose a total order on the entries in an index tree [KF94]. Since all entries are totally ordered by their Hilbert values, the insertion and deletion algorithms are the same as those of the B+-tree except adjusting the MBRs of nodes to cover all descendent MBRs. The Hilbert R-tree was originally proposed to improve the *search* performance of *disk*-resident R-trees. However, here we use the Hilbert value-based ordering to improve the *update* performance of *main-memory* R-trees. Specifically, in the insertion algorithm, the R-tree or the R*-tree examines the MBRs of all nodes encountered to find the node with the least area or overlap enlargement, but the Hilbert R-tree uses a binary search on the total ordering and, therefore, performs only simple value comparisons. In the case of a node overflow, the R-tree or the R*-tree examines all entries in the node and separates them into two groups, but the Hilbert R-tree simply moves half the ordered entries to a new node. In the deletion algorithm, the R-tree or the R*-tree first searches the tree given the MBR of the entry to delete and this search may visit multiple paths. However, the Hilbert R-tree removes an entry with its Hilbert value and does not visit multiple paths. The total ordering in the Hilbert R-tree has another advantage. While the R-tree and R*-tree are non-deterministic in allocating the entries to a node and thus different sequences of insertions result in different tree structures, the Hilbert R-tree does not suffer from such non-determinism.

By applying the QRMBR technique of the CR-tree to R*-tree and Hilbert R-tree, we obtain the cache-conscious R-tree variants **CR*-tree** and **Hilbert CR-tree**, respectively. Their search and update algorithms are the same as those of their non-cache-conscious counterparts except that they use QRMBRs instead of MBRs for search and adjust QRMBRs instead MBRs for update. The QRMBR technique improves the search performance significantly in return for a slight degradation of the update performance caused by the overhead of adjusting QRMBRs.

3.2. Node structures

Fig. 1 shows the node structures of the R-tree variants. C denotes the control information comprising the number of entries in the node and the level of the node in the tree.

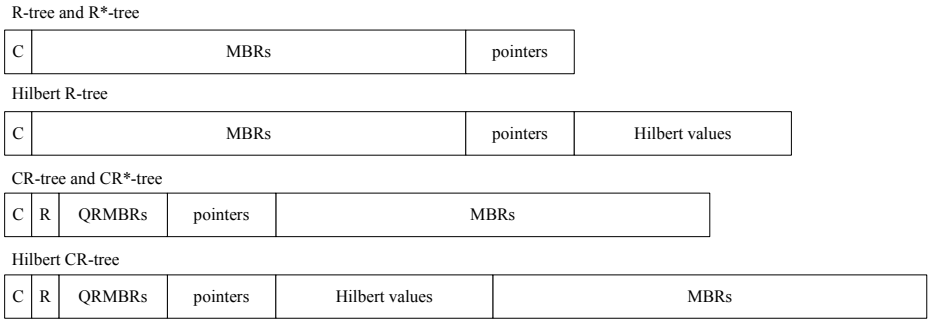


Fig. 1. Node structures of the R-tree variants (C: control information, R: reference MBR)

R denotes the reference MBR used in the QRMBR technique. Each node of the CR-tree, CR*-tree, and Hilbert CR-tree contains uncompressed MBRs corresponding to the QRMBRs to improve the concurrency and update performance. The reason for this is that the QRMBR technique requires re-computing all the QRMBRs in a node when the reference MBR of the node changes. Since the QRMBR technique is a lossy compression scheme, without uncompressed MBRs the recomputation of the QRMBRs requires visiting all children.

The QRMBRs and the Hilbert values make the node size bigger and, therefore, increase the memory consumption. This overhead, however, is not significant. For example, we will see in Table 1 at section 5.1.2 that the largest gap is only 3.2 times between Hilbert CR-trees and R/R*-trees when the node size is 128 bytes. Moreover, increasing the node size does not entail increasing the memory access cost as much. For example, the Hilbert CR-tree, whose node size is the biggest by containing both the QRMBRs and the Hilbert values, reads only the control information, reference MBR, QRMBRs, and pointers for a search operation. Likewise, an update operation reads only the control information, pointers, and Hilbert values before it reaches a leaf node.

4. Concurrency Control of Main-Memory R-trees

4.1. Link technique for R-trees

The OLFIT for main-memory B+-trees improves the performance of concurrent accesses by reducing the coherence cache misses, combined with The B-link technique [CH+01]. For R-trees, We use OLFIT with the GiST-link technique [KMH97]. Like the B-link technique, the GiST-link technique requires all the nodes at each level to be chained together through link pointers. The GiST-link technique uses a node sequence number (NSN) to determine if the right neighbor needs to be examined. The NSN is taken from a counter called the global NSN, which is global in the entire tree and increases monotonically. During a node split, this counter is incremented and the new

```
// Note that QRMBR is used instead of MBR for CR, CR*, and Hilbert CR-trees.
procedure search(query_rectangle)
1.  gnsn:= global_nsn;
2.  push(stack, [root, gnsn]);
3.  while(stack is not empty) {
4.      [node, nsn]= pop(stack);
5.  RETRY:
6.      stack_savepoint = get_savepoint(stack);
7.      result_savepoint:= get_savepoint(result);
8.      saved_version:= node.version;
9.      if (nsn < node.nsn) push(stack, [node.link, nsn]);
10.     if (node is internal) {
11.         gnsn:= global_nsn;
12.         for (each entry [MBR, pointer] in node) {
13.             if (overlaps(query_rectangle, MBR)) push(stack, [pointer, gnsn]);
14.         }
15.     }
16.     else {     // node is a leaf
17.         for (each entry [MBR, pointer] in node) {
18.             if (overlaps(query_rectangle, MBR)) add(result, pointer);
19.         }
20.     }
21.     if (node.latch is locked or node.version ≠ saved_version) {
22.         rollback(stack, stack_savepoint);
23.         rollback (result, result_savepoint);
24.         goto RETRY;
25.     }
26. }
```

Fig. 2. Traversal with the OLFIT for search

value is assigned to the original node. The new sibling node inherits the original node's old NSN. A traversal can now determine whether to follow a link or not by memorizing the global NSN value when reading the parent and comparing it with the NSN of the current node. If the latter is higher, the node must have been split and, therefore, the operation follows the links until it encounters a node whose NSN is less than or equal to the one originally memorized.

In the case of the Hilbert R-tree and the Hilbert CR-tree, we use the GiST-link technique only for the search. We use the B-link technique for the insertion and the deletion because the index entries are totally ordered by their Hilbert values. Link pointers are for dual use as either B-links or GiST-links. In this paper, we consider only the 1-to-2 split for the Hilbert R-tree and the Hilbert CR-tree because the GiST-link technique does not allow redistribution of entries between nodes.

4.2. Search algorithm of R-tree variants using OLFIT

Fig. 2 shows the algorithm for performing R-tree search while using the GiST-link based OLFIT. First, in Lines 1 and 2, it pushes the pointer to the root node and the

global NSN into the stack. Then, in Line 4 it pops the pointer to a node and the associated global NSN from the stack and reads the corresponding node. If the popped node is an internal node, then in Lines 11~14 it pushes into the stack all pointers to the child nodes whose MBRs (or QRMBRs) overlap the query rectangle. If the node is a leaf node, then in Lines 17~19 it adds all pointers to the data objects whose MBRs (or QRMBRs) overlap the query rectangle to the search result. This procedure is repeated until the stack is empty.

Each time it iterates, the pointer to a node is pushed with the value of the global NSN When the pointer to a node in the stack is used to visit a node, in Line 9 the global NSN pushed together is compared with the NSN of the node. If the latter is higher, the node must have been split and, therefore, the link pointer of the node is pushed into the stack together with the original global NSN. This guarantees that the right siblings that split off the original node will also be examined later on.

Lines 5~8 and Lines 21~25 are specific to the OLFIT. Line 8, which saves the version of the node, corresponds to Step R1 of the `ReadNode` primitive in Section 2.2. Line 21, which checks the state of the latch and the version of the node, corresponds to Steps R3 and R4. That is, while reading the node, if the search operation is interfered by other concurrent updates on the node, the condition in Line 21 becomes true and the search operation retries to read the node. (Refer to Line 24 and Line 5). Before the retry, in Lines 22~23 the stack and the result are rolled back to the state recorded in Lines 6~7 before reading the node. The repeatable-read transaction isolation level is achieved by locking all pointers to data objects in the result buffer.

4.3. Update algorithm of R-tree variants using OLFIT

For performing R-tree updates while using the OLFIT concurrency control, the Hilbert R-tree and the Hilbert CR-tree use the B-link technique and the other variants use the GiST-link technique. The insertion operation first looks for the leaf node to hold the new entry, and the deletion operation first looks for the leaf node holding the entry to delete. As in the search, the `ReadNode` primitive presented in Section 2.2 is used in the process. After finding the target leaf, the operations update the leaf node and propagate the update upward using the `UpdateNode` primitive presented in Section 2.2. We omit the detailed algorithm due to space limit. Interested readers are referred to [TR02].

5. Experimental Evaluation

In this section, we compare the index access performance of the main memory R-tree variants with respect to such attributes as data size, data distribution, query selectivity, index node size, the number of parallel threads, and update ratio (= the number of updates / the total number of searches and updates combined). We run our experiments on a Sun Enterprise 5500 server with 8 CPUs (UltraSPARC II, 400MHz) running Solaris 7. Each CPU has 8Mbyte L2 cache whose cache line size is 64 bytes.

5.1. Setup

5.1.1. Data and queries

We use three data sets containing hundred thousand (100K), one million (1M), and ten million (10M) data rectangles each. All rectangles have the same size 4m × 4m, and their centers are either uniformly distributed or skewed within a 40km × 40km region. Skewed data are simulated with the Gaussian distribution of mean 20,000m and standard deviation 200m.

We use two region queries, where the regions are specified with square windows whose centers are distributed uniformly within the 40km × 40km region. The window sizes are 126m × 126m and 400m × 400m and the resulting selectivity values are 0.001% and 0.01%, respectively.

Updates are performed as a sequence of random moves. Each move deletes an entry at the coordinates <x, y> and inserts it into a new position at $<x \pm 30r_1, y \pm 30r_2>$ where r_1 and r_2, $0 \leq r_1, r_2 \leq 1$, are random numbers. This random move is one of the cases that can be generated using the GSTD software [TN00] and is typical of moving objects. Assuming cars are moving at 100km/hour (= 30m/second), we choose 30m for the variation. The variation does not affect the update performance significantly because the update operation consists of independent two operations, delete and insert.

5.1.2. Indexes

We implement the R-tree and its five variants R*-tree, Hilbert R-tree, CR-tree, CR*-tree, and Hilbert CR-tree. (In this section, we label them as R, R*, HR, CR, CR*, and HCR) We allow duplicate key values in these indexes and initialize them by inserting data rectangles and the associated pointers. We set the pointer size to 4bytes as we run our experiment in the 32-bit addressing mode. Additionally, we use 4-byte QRMBRs in the CR, CR*, and Hilbert CR-trees. In the R*-tree and the CR*-tree, we do not use the reinsertion because it makes the concurrency control difficult without latching the whole tree, nor we use the least overlap enlargement policy because it improves the search performance only slightly at the expense of significant update performance.

Table 1 shows the node fanout, index height, and index size of the main-memory

Table 1. Node fanout, Index height and Index size for different node sizes (data size=1M, uniform dist.)

Node size (bytes)	Fanout						Height						Index size (Mbytes)					
	R	R*	HR	CR	CR*	HCR	R	R*	HR	CR	CR*	HCR	R	R*	HR	CR	CR*	HCR
128	5	5	3	4	4	3	11	11	34	13	13	34	50	50	149	65	65	159
256	11	11	8	9	9	7	7	7	8	8	8	9	39	39	54	50	50	65
384	18	18	13	15	15	11	6	6	7	6	6	7	35	34	49	42	41	57
512	24	24	17	20	20	15	5	5	6	6	6	6	34	33	47	41	40	54
1024	50	50	35	41	41	31	4	4	5	5	5	5	32	31	44	39	38	50
2048	101	101	72	84	84	63	4	4	4	4	4	4	31	30	42	37	36	48
3072	152	152	109	127	127	95	3	3	4	4	4	4	31	30	42	37	36	47

R-tree variants created on the data of size 1M. The numbers are based on the index entry size 20 bytes for the R and R*-tree, 24 bytes for the CR and CR*-tree, 28 bytes for the Hilbert R-tree, and 32 bytes for the Hilbert CR-tree.

5.1.3. Experimental outline

We perform two kinds of experiments: the sequential access experiment and the concurrent access experiment. In each experiment, we measure the throughput of a sequence of search (i.e., range query) and update (i.e., random move) operations. Operation/sec refers to the number of executed operations divided by the total execution time.

In the sequential access experiment, we initialize indexes by inserting data objects in sequence and perform searches using a sequence of region queries mentioned above. In the concurrent access experiment, we initialize indexes inserting data objects concurrently in eight threads and then compare the concurrent search and update performance between the OLFIT and the conventional latched-based link technique [LY81, KMH97]. The performed searches and updates are the same as those in the sequential access experiment and are divided evenly to each thread. We also compare the performance among OLFIT techniques for a mixture of searches and updates mixed at different ratios.

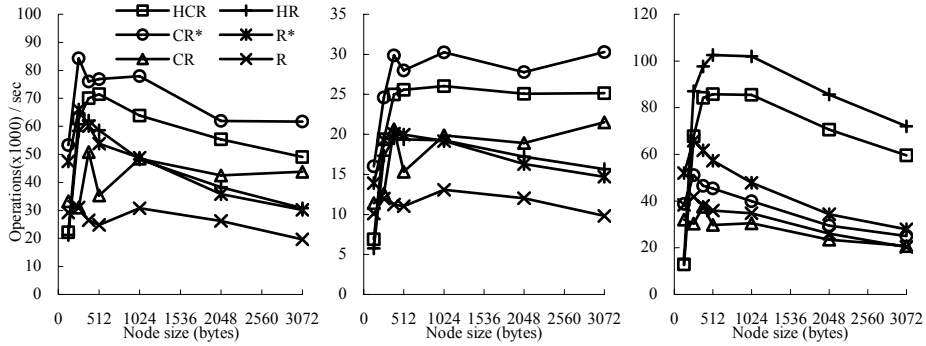
We omit such a mixture of operations in the sequential experiment because the resulting performance is a linear interpolation between the results from searches only and updates only and, thus, is calculated without running the actual experiment. In the concurrent case, the performance from the mixture is not an interpolation because search and update may interfere with each other.

The way QRMBR and OLFIT techniques improve the throughput is by reducing the number of L2 cache misses. There are, however, other factors contributing to improving the throughput as well, like the number of instructions in the code. Since main memory performance is particularly sensitive to code implementations, serious attention should be paid to removing code-specific biases among the R-tree variants. In this regard, we demonstrate the consistency between the performance measured as the throughput and the performance measured as the number of cache misses (using the *Perfmon* tool [Enb99]).

There is no R-tree variant winning consistently in all possible cases. Therefore, it could be misleading to rank the variants by their search or update performance without considering the complexity of the comparisons. In this regard, we have performed a benchmark of 288 test cases generated as a combination of the attributes mentioned above and selected the winners by their rates of winning the cases. The results obtained are consistent with those obtained in the experiments presented in this section. We omit the results due to space limit. Interested readers are referred to [TR02].

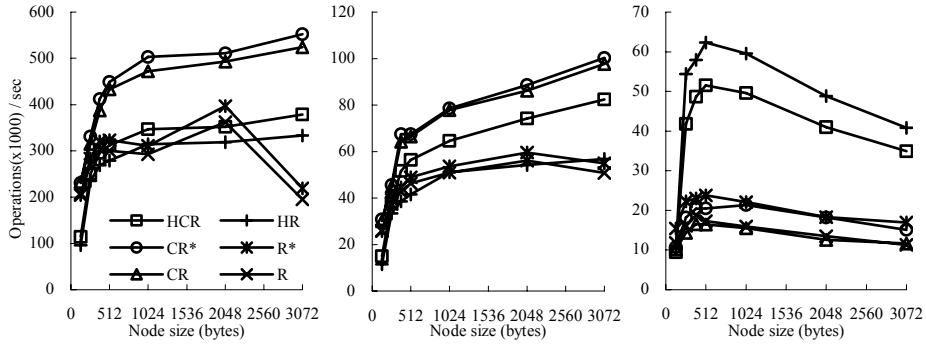
5.2. Sequential access performance

Fig. 3 shows the sequential access performance for different node sizes when one million data rectangles (1M) are uniformly distributed, and Fig. 4 shows the same infor-



(a) Search (selectivity=0.001%) (b) Search (selectivity=0.01%) (c) Update

Fig. 3. Sequential access performance w.r.to node size (data size = 1M, uniform dist.)

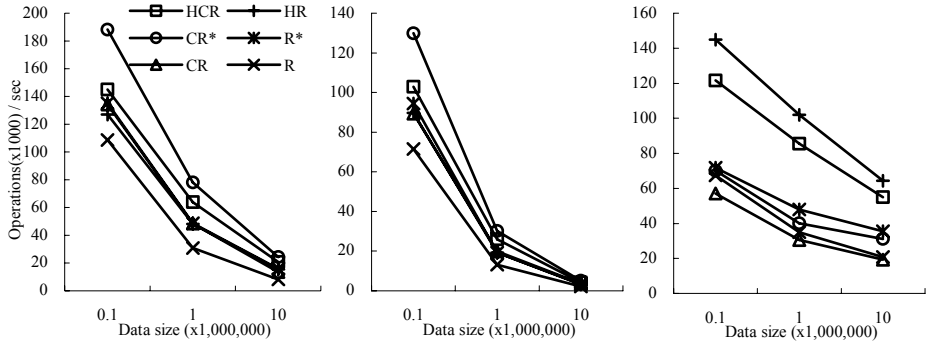


(a) Search (selectivity=0.001%) (b) Search (selectivity=0.01%) (c) Update

Fig. 4. Sequential access performance w.r.to node size (data size=1M, Gaussian dist.)

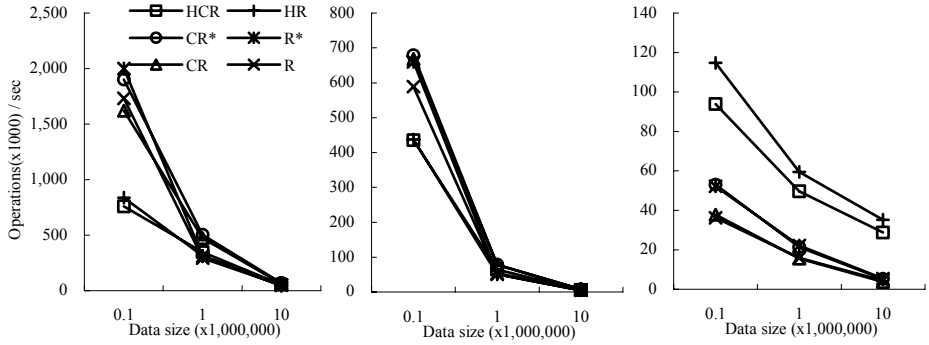
mation when the data rectangles have the Gaussian distribution. From Fig. 3(a)-(b) and Fig. 4(a)-(b), we make the following observations about the search performance for both data distributions alike. First, they confirm that the QRMBR technique improves the search performance significantly. That is, a cache-conscious version (i.e., CR, CR*, HCR) is better than its non-cache-conscious counterpart (i.e., R, R*, HR). Second, CR*-trees show the best search performance, which is attributed to not only the QRMBR technique but also the well-clustered nodes generated with the R*-tree split algorithm. Third, the search performance fluctuates as the index node size increases. This reflects the dual effect of increasing the fanout – it increases the cost of reading a node but decrease the overall search cost by reducing the tree height.

From the same figures, we see that Hilbert CR-trees perform better than CR-trees in uniformly distributed data but worse in skewed data. That is, skewed data gives a disadvantage to Hilbert CR-trees. This result contradicts Kamel and Faloutsos's conjecture [KF94] that skewness of data gives a favor to Hilbert R-trees. The cause is the difference in the experimental settings. For instance, their experiment uses 2-to-3 split in Hilbert R-trees and reinsertion in R*-trees whereas ours does not.



(a) Search (selectivity=0.001%) (b) Search (selectivity=0.01%) (c) Update

Fig. 5. Sequential access performance w.r.to data size (node size=1024B, uniform dist.)



(a) Search (selectivity=0.001%) (b) Search (selectivity=0.01%) (c) Update

Fig. 6. Sequential access performance w.r.to data size (node size=1024B, Gaussian dist.)

From Fig. 3(c) and Fig. 4(c), we make the following observations about the update performance for both data distributions. First, they confirm that the Hilbert value-based ordering improves the update performance significantly. That is, Hilbert R-trees are better than R-trees and R*-trees, and Hilbert CR-trees are better than CR-trees and CR*-trees. Second, the update performance of Hilbert R-trees is better than that of Hilbert CR-trees. This is due to the Hilbert CR-tree's overhead of maintaining the QRMBRs. Third, CR*-trees show poor update performance, unlike their excellent search performance. This is due to the computational overhead of R*-tree insertion for finding the leaf node with the minimum overlap enlargement and splitting a node.

Fig. 5 and Fig. 6 show the search and update performance with respect to the size of data with each of the two distributions. First, it appears that the performance gap among the R-tree variants decreases as the data size increases. This is true for the absolute performance, but it is the opposite for the relative performance. The reason for this increase is that the increase of data size causes more cache misses and consequentially highlights the performance gain of the QRMBR.

Second, with uniformly distributed data, the performance rank among the R-tree variants is the same for all data sizes whereas, with the skewed data, the cache-conscious versions lose advantage in the search performance as the data size decreases. This happens partly due to the computation overhead of the QRMBR method for reducing the number of cache misses. In addition, three factors reduce the effectiveness of the QRMBR. First, if the data is small enough to fit in the cache, a cache miss hardly occurs and, therefore, there is little gain from the QRMBR. Second, data skew reduces the gap between the size of a parent node's MBR and the size of its child node's MBR, and this diminishes the effectiveness of the relative representation of an MBR and, as a result, increases the quantization errors. Third, these quantization errors are higher for lower query selectivity. The aforementioned instance in Fig. 6(a) is the worst case caused by the accumulation of these three factors.

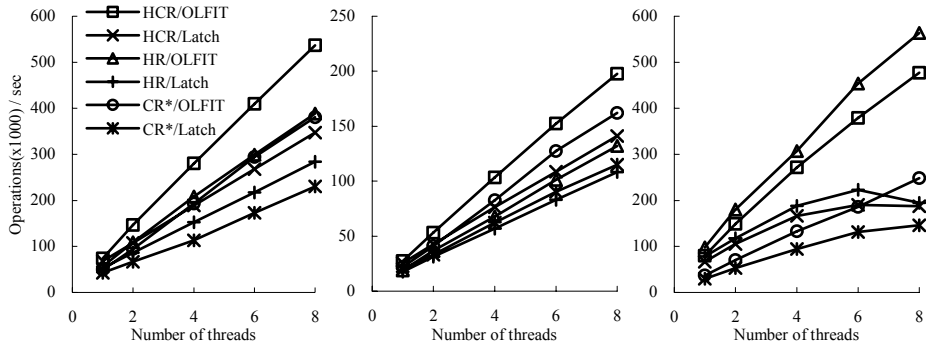
From all these observations about sequential access performance, we judge that Hilbert CR-trees are the best considering both the search performance and the update performance. These trees are not the first choice in any category, but are consistently the second or the third choice by a small margin in most cases. In summary, we conclude that CR*-trees are the best choice for the search performance only, Hilbert R-trees are the best choice for the update performance only, and Hilbert CR-trees are the best choice when considering both.

5.3. Concurrent access performance

Fig. 7 shows the search and update performance for different numbers of threads, contrasted between the conventional latch-based and the OLFIT-based concurrency control, given the data with size 1M and the uniform distribution. (We omit the results obtained with the Gaussian distribution data due to space limit. Most observations are same as in the uniform distribution case.) We set the node size to 512 bytes, which is the median of the seven different node sizes used. Besides, we consider only the CR*-tree, Hilbert R-tree, and Hilbert CR-tree because the other three variants are poorer in both the search and update performance.

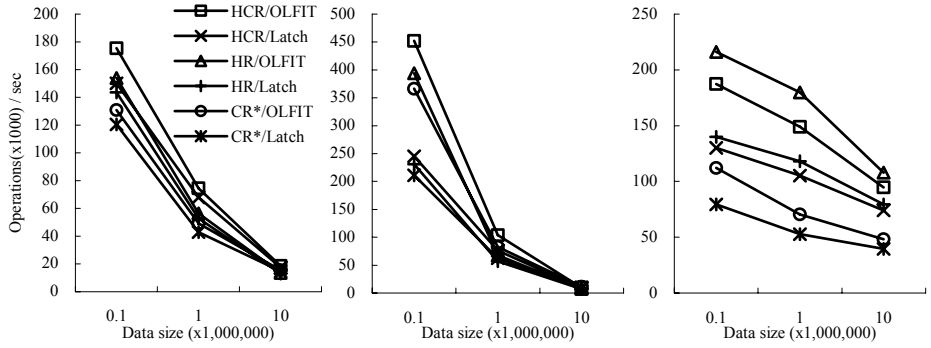
From this figure, we make the following observations. First, they confirm the advantage of the OLFIT in the concurrent search and update performance. That is, as the number of threads increases, CR*-trees, Hilbert R-trees, and Hilbert CR-trees become significantly better with the OLFIT than with the Latch. Second, the relative search performance among the R-tree variants differs between the two data distributions. Specifically, the best search performer is Hilbert CR-trees for the uniform distribution and CR*-trees for the Gaussian distribution. The reason is that data skew is not favorable to Hilbert R-trees, as discussed in the sequential access case**.** Third, the update performance shows the same relative performance as in the sequential access experiment.

Fig. 8 shows the concurrent search and update performance with respect to the data size for the uniform distribution given the number of threads 4. (We omit the results from the Gaussian distribution and other number of threads for the same reason as above.) We make the following observations. First, like the case of sequential access



(a) Search (selectivity=0.001%) (b) Search (selectivity=0.01%) (c) Update

Fig. 7. Concurrent access performance w.r.to the number of threads (data size=1M, uniform)

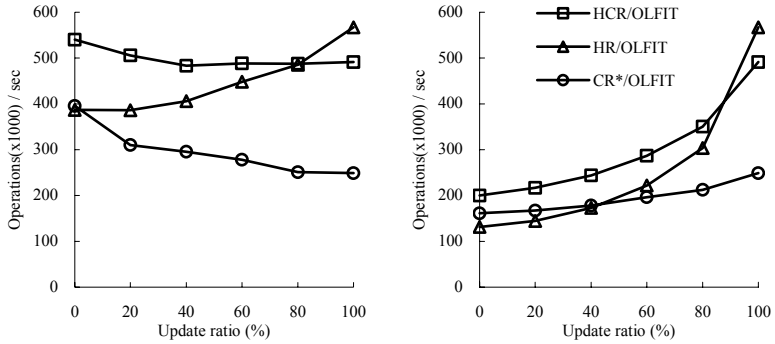


(a) Search (selectivity=0.001%) (b) Search (selectivity=0.01%) (c) Update

Fig. 8. Concurrent access performance w.r.to data size (4 threads, uniform dist.)

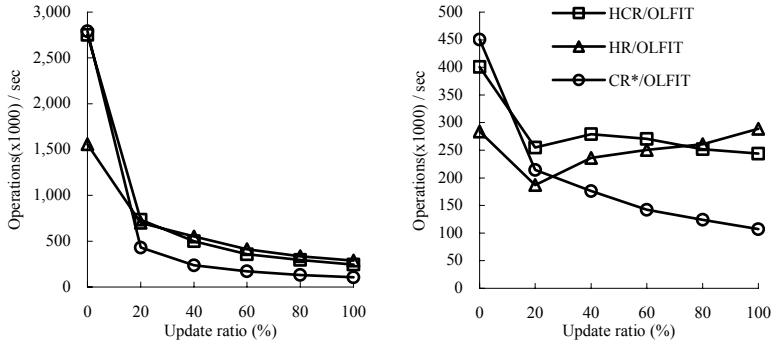
performance, the absolute performance gap decreases among the R-tree variants while the relative performance gap increases as the data size increases. Second, the performance advantage of the OLFIT over the Latch becomes more noticeable for smaller data and queries with lower selectivity. We draw the following reasons for this. First, smaller data size increases the coherence cache miss rate because evidently it increases the possibility of cached data being invalidated by another processor. Second, in the case of higher query selectivity, queries access nodes near the leaves in addition to those near the root. This causes other types of cache misses (e.g., capacity cache miss) to occur as well and, as a result, reduces the relative adverse effect of the coherence cache misses.

Fig. 9 and Fig. 10 show the concurrent access performance for different update ratios for each of the two data distributions**.** The OLFIT is used for the concurrency control and the number of threads is fixed to eight. We make the following observations from these figures. First, the winners change places as the update ratio changes. The pattern is slightly different between the two data distributions. In the case of the uniform distribution, Hilbert CR-trees are the best in the low to middle range of the



(a) Search selectivity = 0.001% (b) Search selectivity = 0.01%

Fig. 9. Concurrent access performance w.r.to update ratios (8 threads, data size=1M, uniform)

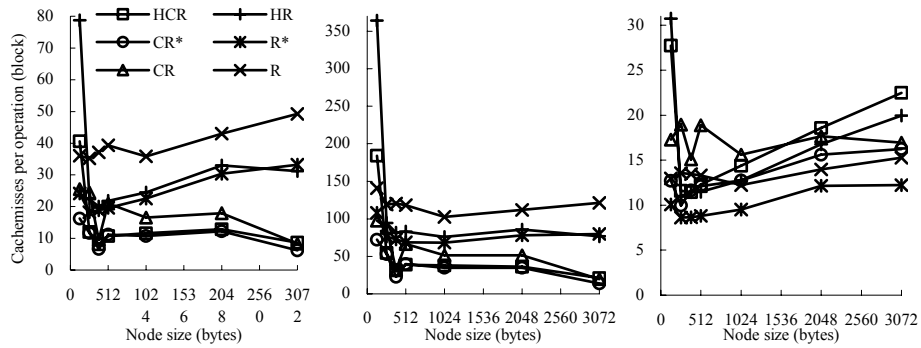


(a) Search selectivity = 0.001% (b) Search selectivity = 0.01%

Fig. 10. Concurrent access performance w.r.to update ratios (8 threads, data size=1M,Gaussian)

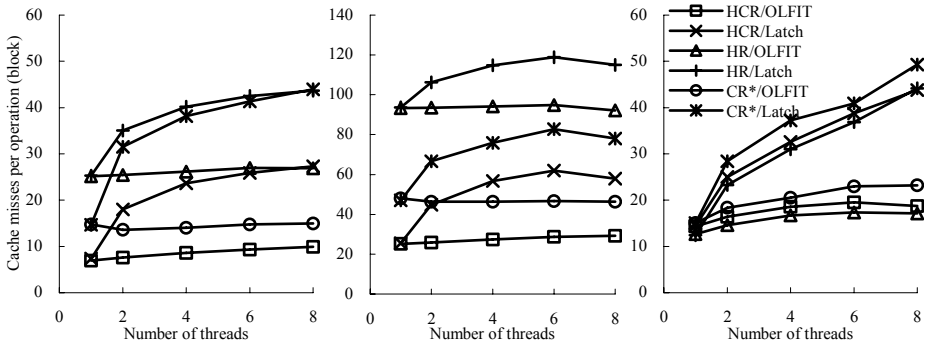
update ratio and Hilbert R-trees are the best in the high range. Hilbert CR-trees fall below Hilbert R-trees as the cost of managing QRMBR increases. In the case of the Gaussian distribution, CR*-trees are the best or comparable to Hilbert CR-trees in the low range, Hilbert CR-trees are the best in the middle range, and Hilbert R-trees are the best in the high range. The initial lead of CR*-trees is due to the relatively poor search performance of Hilbert R-trees and Hilbert CR-trees against skewed data. CR*-trees fall below the other two as the number of updates increases due to the increasing computational overhead. Second, related to the first observation, Hilbert CR-trees have an advantage over the other two trees at the higher query selectivity.

From all these observations about concurrent access performance, we make the same judgment as in the sequential access performance. Hilbert CR-trees are the best choice for the search performance if the data is distributed uniformly whereas CR*-trees are the best if the data is skewed, Hilbert R-trees are the best choice for the concurrent update performance, and Hilbert CR-trees are the best choice when considering both.



(a) Search (selectivity=0.001%) (b) Search (selectivity=0.01%) (c) Update

Fig. 11. The number of cache misses (Sequential access, data size=1M, uniform)



(a) Search (selectivity=0.001%) (b) Search (selectivity=0.01%) (c) Update

Fig. 12. The number of cache misses (Concurrent access, data size=1M, uniform)

5.4. Consistency with the number of cache misses

Fig. 11 shows the number of cache misses in sequential accesses for different node sizes when one million data rectangles are uniformly distributed, and Fig. 12 shows the number in concurrent accesses for different numbers of threads. We do not show the case of skewed data because the observed results are the same.

The numbers of cache misses of the R-tree variants in Fig. 11(a)-(b) are ranked exactly in the reverse order of the throughputs in Fig. 3(a)-(b), and the numbers in Fig. 11(c) are almost in the reverse order of those in Fig. 3(c). In Fig. 11(c), the HCR-tree and the HR-tree incur more cache misses than the other R-tree variants despite showing the best update performance in Fig. 3(c). This is because an insertion in HCR-trees and HR-trees needs far less computation than the other R-tree variants for choosing the appropriate leaf node. Fig. 12 shows that the number of cache misses of the variants are ranked in the reverse order of the throughputs in Fig. 7(a)-(c), including the update case. We see that the numbers hardly increase with the number of threads

if OLFIT is used, whereas they do increase if Latch is used. This confirms the advantage of OLFIT over Latch in concurrency control.

6. Summary and Further Work

In this paper, we compared the sequential and concurrent access (search and update) performance of the main-memory R-tree and its five variants – R*-tree, Hilbert R-tree, CR-tree, CR*-tree, and Hilbert CR-tree – while applying the QRMBR technique for faster search performance and the OLFIT technique for better concurrency control. We used the GiST-link technique to apply the OLFIT technique to the R-tree variants. Naturally, the QRMBR improved the index search performance and the OLFIT improved the concurrency.

We conducted experiments for evaluating the performance in terms of the throughput. As a result, we found the following trees performing the best in each category: in sequential accesses, CR*-trees for search, Hilbert R-trees for update, and Hilbert CR-trees when considering both and, in concurrent accesses, Hilbert CR-trees for searching uniformly distributed data, CR*-trees for searching skewed data, Hilbert R-trees for update, and Hilbert CR-tree for a mixture of search and update except an update-intensive case. We also demonstrated that the throughput results were not biased by the code implementations by showing the consistency between the observations based on the number of cache misses and those based on the throughput.

We also demonstrated that the throughput results were not biased by the code implementation by showing the consistency between the observations based on the number of cache misses and those based on the throughput.

All queries considered in this paper are range search queries. We plan to pursue further experiments using the nearest neighbor queries. Like range search queries, these queries involve traversing R-trees while pushing nodes into and popping nodes from a stack. Therefore, they are amenable to tree structures clustered so that more index entries can be examined with the same block cached. Compared with range queries, however, nearest neighbor queries incur higher computation cost and, therefore, clustering is more important than compression to search performance.

Currently our R-tree search algorithm does not prevent the phantom phenomenon. To our knowledge, there does not exist any algorithm addressing this problem for main-memory indexes. We are currently working on it.

Other further works include using real (dynamic) geographical data sets instead of synthetic ones and using different process architecture like MPP instead of SMP. As MPP incurs higher communication cost among processors than SMP, we expect the advantage of the OLFIT over the latch-based concurrency control should become eminent. We also plan to incorporate the R-tree variants into a main memory spatial data management system and perform a benchmark comparison instead of simulation.

7. References

[BMR01] Philip Bohannon, Peter McIlroy, and Rajeev Rastogi, "Improving Main-Memory Index Performance with Partial Key Information", In *Proc. of ACM SIGMOD Conf.*, 2001, pages 163-174.

[BS77] Rudolf Bayer and Mario Schkolnick, "Concurrency of Operations on B-Trees", *Acta Informatica 9*, 1977, pages 1-21.

[BK+90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", In *Proc. of ACM SIGMOD Conf.*, 1990, pages 322-331.

[CGM01] Shimin Chen, Philip B. Gibbons, and Todd C. Mowry, "Improving Index Performance through Prefetching", In *Proc. of ACM SIGMOD Conf.*, 2001, pages 235-246.

[CH+01] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon, "Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems", In *Proc. of VLDB Conf.*, 2001

[Enb99] R. Enbody, Perfmon Performance Monitoring Tool, 1999, available from http://www.cps.msu.edu/~enbody/perfmon.html.

[Gut84] Antonin Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching", In *Proc. of ACM SIGMOD Conf.*, 1984, pages 125-135.

[JJ+01] Ravi Jain, Christian S. Jensen, Ralf-Hartmut Güting, Andreas Reuter, Evaggelia Pitoura, Ouri Wolfson, George Samaras, and Rainer Malaka, "Managing location information for billions of gizmos on the move—what's in it for the database folks?", *IEEE ICDE 2001 Panel*.

[KCK01] Kihong Kim, Sang K. Cha, and Keunjoo Kwon, "Optimizing Multidimensional Index Trees for Main Memory Access", In *Proc. of ACM SIGMOD Conf.*, 2001, pages 139-150.

[KF94] Ibrahim Kamel and Christos Faloutsos, "Hilbert R-tree: An Improved R-tree using Fractals", In *Proc. of VLDB Conf.*, 1994, pages 500-509.

[KMH97] Marcel Kornacker, C. Mohan, and Joseph M. Hellerstein, "Concurrency and Recovery in Generalized Search Trees", In *Proc. of ACM SIGMOD Conf.*, 1997, pages 62-72.

[LKC01] Juchang Lee, Kihong Kim, and Sang K. Cha, "Differential Logging: A Commutative and Associative Logging Scheme for Highly Parallel Main Memory Database", In *Proc. of IEEE ICDE Conf.*, 2001, pages 173-182.

[LY81] Philip L. Lehman and S. Bing Yao, "Efficient Locking for Concurrent Operations on B-Trees", *ACM TODS*, Vol. 6, No. 4, 1981, pages 650-670.

[RR00] Jun Rao and Kenneth Ross, "Making B+-trees Cache Conscious in Main Memory", In *Proc. of ACM SIGMOD Conf.*, 2000, pages 475-486.

[SY+02] Yasushi Sakurai, Masatoshi Yoshikawa, Shunsuke Uemura, Haruhiko Kojima, "Spatial Index of High-dimensional Data Based on Relative Approximation", *VLDB Journal* 11(2), 2002, pages 93-108

[TN00] Y. Theodoridis and M.A. Nascimento, "Generating Spatio temporal Datasets on the WWW", *ACM SIGMOD Record*, September 2000.

[TR02] Sangyong Hwang, Keunjoo Kwon, Sang K. Cha, Byung S. Lee, "Performance Evaluation of Main-Memory R-tree Variants", *Technical Report*, 2002, available at http://kdb.snu.ac.kr/papers/SSTD03_TR.pdf