Optimizing Multidimensional Index Trees for Main Memory Access

Kihong Kim Sang K. Cha Keunjoo Kwon
School of Electrical Engineering and Computer Science
Seoul National University
{ next, chask, icdi }@kdb.snu.ac.kr

Abstract

Recent studies have shown that cache-conscious indexes such as the CSB+-tree outperform conventional main memory indexes such as the T-tree. The key idea of these cache-conscious indexes is to eliminate most of child pointers from a node to increase the fanout of the tree. When the node size is chosen in the order of the cache block size, this pointer elimination effectively reduces the tree height, and thus improves the cache behavior of the index. However, the pointer elimination cannot be directly applied to multidimensional index structures such as the R-tree, where the size of a key, typically, an MBR (minimum bounding rectangle), is much larger than that of a pointer. Simple elimination of four-byte pointers does not help much to pack more entries in a node.

This paper proposes a cache-conscious version of the R-tree called the CR-tree. To pack more entries in a node, the CR-tree compresses MBR keys, which occupy almost 80% of index data in the two-dimensional case. It first represents the coordinates of an MBR key relatively to the lower left corner of its parent MBR to eliminate the leading 0's from the relative coordinate representation. Then, it quantizes the relative coordinates with a fixed number of bits to further cut off the trailing less significant bits. Consequently, the CR-tree becomes significantly wider and smaller than the ordinary R-tree. Our experimental and analytical study shows that the two-dimensional CR-tree performs search up to 2.5 times faster than the ordinary R-tree while maintaining similar update performance and consuming about 60% less memory space.

1. Introduction

As the price of memory continues to drop below $1,000/GB, it is now feasible to place many of the database tables and indexes in main memory. With such memory-resident tables and indexes, the traditional bottleneck of disk access almost disappears, especially for search transactions. Instead, memory access becomes a new bottleneck [1]. A recent study with commercial DBMSs has shown that half the execution time is spent on memory access when the whole database fits in memory [2]. Since the speed in DRAM chips has been traded off for the capacity, the gap between the CPU speed and the DRAM speed has grown significantly during the past decade [3]. In today's computer systems, each memory access costs tens of processor cycles. To overcome this gap, modern processors adopt up to several megabytes of SRAM as the cache, which can be accessed in just one or two processor cycles.

Recognizing the widening gap between the CPU speed and the DRAM speed, Rao and Ross recently addressed the importance of the cache behavior in the design of main memory indexes and showed that the cache-conscious search tree (CSS-tree) performs lookups much faster than the binary search tree and the T-tree in the read-only OLAP environment [4]. They also observed the reasonably good cache behavior of the B+-tree and proposed its cache sensitive variants [5]. Called CSB+-tree, these B+-tree variants store child nodes contiguously in memory to eliminate most child pointers except the first one. The location of the *i*-th child node is computed from that of the first child. Providing more room for keys in the node, this pointer elimination approach effectively doubles the fanout of the B+-tree. Given the node size in the order of the cache block size, the fanout doubling reduces the height of B+-tree, and thus incurs less cache misses during the tree traversal than the B+-tree. Note that such a pointer elimination technique does not provide much benefit in disk-resident indexes where the fanout is typically a few hundreds and doubling the fanout does not lead to the immediate reduction in the tree height (e.g., $\lceil \log_{200} 10^6 \rceil = \lceil \log_{400} 10^6 \rceil = 3$).

The pointer elimination technique cannot be directly applied to multidimensional index structures such as the R-tree [6], because multidimensional keys, typically, MBRs (minimum bounding rectangles), are much larger than pointers. Thus, pointer elimination alone cannot widen the index tree significantly. For example, when the 16-byte MBR is used for the two-dimensional key, the simple elimination of a 4-byte pointer provides at most 25% more room for the keys, and this increase is not big enough to make significant difference in the tree height for the improved cache behavior.

Recognizing that MBR keys occupy most of index data in the multidimensional index, for example, almost 80% for the 2D R-tree, this paper focuses on inexpensive compression of MBR keys to improve the index cache behavior. Called CR-Tree (Cache-conscious R-Tree), it takes advantage of the fact that the child nodes are grouped into a parent node such that each node occupies a small portion of the data space of its parent node [6][7][8]. Thus, if we represent an MBR relatively to its parent MBR, the coordinates of the resultant relative MBR have a fewer number of significant bits with many leading 0's. To further reduce the number of bits per MBR, the CR-tree cuts off trailing insignificant bits by quantization. Our analysis and experiment show that this

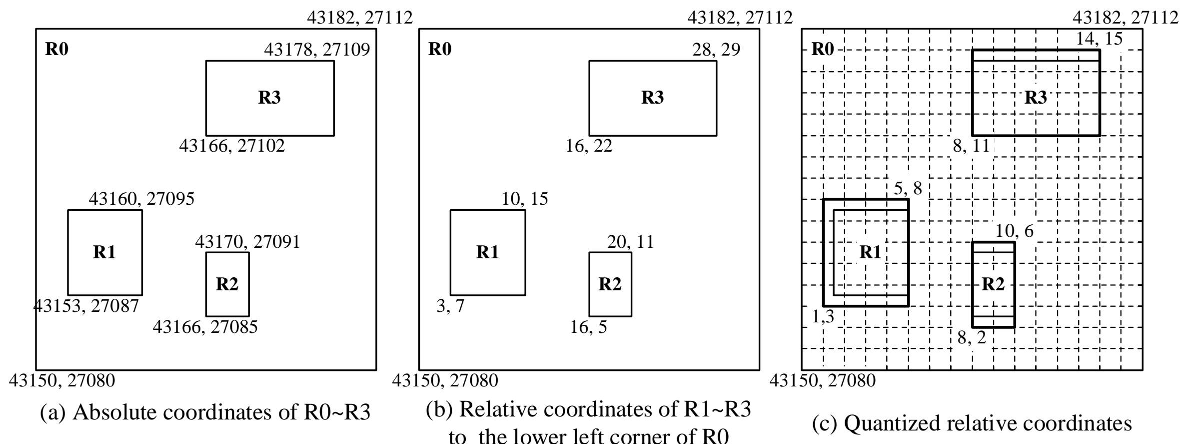

Figure 1: QRMBR Technique

compression technique can reduce the MBR size to less than a fourth, thereby increasing the fanout by more than 150%. A potential problem with the proposed technique is that the information loss by quantization may increase false hits, which have to be filtered out through a subsequent refinement step in most multidimensional indexes [9]. However, we can keep the number of false hits negligibly small by the proper choice of the quantization level so that the cost of filtering out false hits can be paid off by the significant savings in cache misses.

This paper also explores several options in the design of CR-tree including whether to use the pointer elimination technique of the CSB+-tree, whether to apply the proposed compression technique to leaf nodes or not, the choice of quantization levels, and the choice of node size. Our experimental study shows that all the resultant CR-tree variants significantly outperform the R-tree in terms of the search performance and the space requirement. The basic CR-tree that uses only the proposed technique performs search operations up to 2.5 times faster than the R-tree while performing update operations similarly to the R-tree and using about 54% less memory space. Compared with the basic CR-tree, most of CR-tree variants use less memory with algorithmic overhead. Our analysis of the proposed technique and various indexes used in our experiment coincides with the experimental result.

This paper is organized as follows. Section 2 presents the basic idea of this paper and formulates our problem. Section 3 presents the proposed MBR compression scheme, and the section 4 describes the proposed CR-tree. Section 5 analytically compares the CR-tree with the ordinary R-tree, and section 6 presents the result of the experiment conducted to compare the CR-tree with the R-tree. Section 7 finally concludes this paper.

	L1 Cache	L2 Cache	Memory
Block size	16~32B	32~64B	4~16KB
Size	16~64KB	256KB~8MB	~32GB
Hit time	1 clock cycle	1~4 clock cycles	10~40 clock cycles
Backing store	L2 cache	Memory	Disks
Miss penalty	4~20 clock cycles	40~200 clock cycles	~6M clock cycles

Table 1: Summary of Current Memory Hierarchy

2. Motivation

2.1 Memory Hierarchy

Table 1 summarizes the properties of the memory hierarchy observed in Sun UltraSPARC II and Intel Xeon platforms. In UltraSPARC II, the block size is 32 bytes for the L1 cache and 64 bytes for the L2 cache [10]. Typically, the L1 cache can be accessed in one clock cycle, and the L2 cache can be accessed in two clock cycles. The memory access time depends on the DRAM type. When EDO DRAM is used, each memory access takes 50 ns on average. When a cache miss occurs in the L1 cache and the L2 cache, a victim is selected. The miss penalty is the cost of selecting a victim and accessing the backing store. In UltraSPARC II, each L1 cache miss incurs two accesses to the L2 cache, and each L2 cache miss incurs four accesses to main memory.

2.2 Basic Idea

The idea in this paper is to make the R-tree cache-conscious by compressing MBRs. Figure 1 illustrates the compression scheme used in this paper. Figure 1(a) shows the absolute coordinates of $R0$~$R3$. Figure 1(b) shows the coordinates of $R1$~$R3$ represented relatively to the lower left corner of $R0$. These relative coordinates have a less number of significant bits than absolute coordinates. Figure 1(c) shows the coordinates of $R1$~$R3$ quantized into 16 levels or four bits by cutting off trailing insignificant bits. We call the resultant MBR QRMBR (quantized relative representation of MBR). Note that QRMBRs can be slightly bigger than original MBRs.

The CR-tree is a cache-conscious R-tree that uses QRMBRs as index keys. For the sake of simplicity, the quantization levels are made the same for all nodes. Figure 2 shows the structure of a CR-tree node that can contain up to M entries. It keeps a flag indicating whether it is a leaf or not, the number of stored entries, and the reference MBR that tightly encloses its entire child MBRs. The reference MBR is used to calculate the QRMBRs stored in the node. Internal nodes store entries of the form (*QRMBR*, *ptr*), where *ptr* is the address of a child node and *QRMBR* is a quantized relative representation of the child node MBR. Leaf nodes store entries of the form (*QRMBR*, *ptr*), where *ptr* refers to an object and *QRMBR* is a quantized relative representation of the

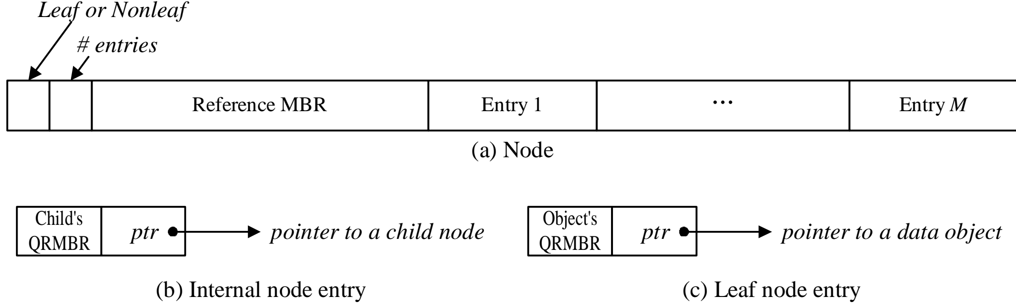

Figure 2: Data Structure of the CR-tree

object MBR. In most of our experiments, we quantize each of x and y coordinates into 256 levels or one byte.

2.3 Problem Formulation

Our goal is to reduce the multidimensional index search time in main memory databases.

Observation 1. Let c be the node size in the number of cache blocks, and $N_{node\ access}$ be the number of nodes accessed during search. The main memory indexes need to be designed to minimize $c \cdot N_{node\ access}$.

In main memory, the index search time mainly consists of the key comparison time and the memory access time incurred by cache misses. If a cache miss occurs, the CPU has to wait until the missing data are cached. A cache miss can occur for three reasons: missing data, missing instructions, and missing TLB (table look-aside buffer) entries. Therefore, we can roughly express our goal as minimizing

$$T_{index\ search} \cong T_{key\ compare} + T_{data\ cache} + T_{TLB\ cache}$$

where $T_{key\ compare}$ is the time spent comparing cached keys, $T_{data\ cache}$ is the time spent caching data, and $T_{TLB\ cache}$ is the time spent caching TLB entries. For simplicity, we omit the time for caching missing instructions because the number of instruction misses mostly depends on the compiler and we can hardly control it.

Let $C_{key\ compare}$ be the key comparison cost per cache block, $C_{cache\ miss}$ be the cost of handling a single cache miss, and $C_{TLB\ miss}$ be the cost of handling a single TLB miss. When the node size is smaller than that of a memory page, each access to a node incurs at most one TLB miss. For simplicity, we assume that nodes have been allocated randomly and that no node and no TLB entry are cached initially. Then,

$$\begin{aligned} T_{index\ search} &= c \cdot C_{key\ compare} \cdot N_{node\ access} \\ &\quad + c \cdot C_{cache\ miss} \cdot N_{node\ access} \\ &\quad + C_{TLB\ miss} \cdot N_{node\ access} \\ &= c \cdot N_{node\ access} \cdot (C_{key\ compare} + C_{cache\ miss} + C_{TLB\ miss} / c) \end{aligned}$$

Since $C_{cache\ miss}$ and $C_{TLB\ miss}$ are constant for a given platform, we can control three parameters: c, $C_{key\ compare}$, and $N_{node\ access}$. Among them, we cannot expect to reduce $C_{key\ compare}$ noticeably because the key comparison is generally very simple. In addition, $C_{TLB\ miss}$ and $C_{cache\ miss}$ typically have similar values. Therefore, the index search time mostly depends on $c \cdot N_{node\ access}$.

Observation 2. The amount of accessed index data can be best reduced by compressing index entries.

The term $c \cdot N_{node\ access}$ can be minimized in three ways: changing the node size such that $c \cdot N_{node\ access}$ becomes minimal, packing more entries into a fixed-size node, and clustering index entries into nodes efficiently. The second is often termed as compression and the third as clustering [11].

The optimal node size is equal to the cache block size in one-dimensional case. In one-dimensional trees such as the B+-tree, since exactly one internal node is accessed for each level, the number of visited internal nodes decreases logarithmically the node size. On the other hand, the number of visited leaf nodes decreases linearly with the node size, and c increases linearly with the node size. Therefore, $c \cdot N_{node\ access}$ increases with the node size, and thus it is minimal when c is one.

In multidimensional indexes, more than one internal nodes of the same level can be accessed even for the exact match query, and the number of accessed nodes of the same level decreases as the node size increases. Since this decrease is combined with the log scale decrease of tree height, there is a possibility that the combined decrease rate of node accesses exceeds the linear increase rate of c. We will show analytically in section 5.2 that the optimal node size depends on several factors like the query selectivity and the cardinality.

Compressing index entries is equivalent to increasing the node size without increasing c. In other words, it reduces $N_{node\ access}$ while keeping c fixed. Thus, it is highly desirable. Compression has been addressed frequently in disk-based indexes because it can reduce the tree height, but there is little dedicated work, especially in multidimensional indexes. The following simple analysis shows that why the compression in disk-resident indexes does not provide as significant gain as in main memory indexes.

Suppose that the tree A can pack f entries on average in a node and the tree B can pack $2f$ entries in a node using a good compression scheme. Then, their expected height is $\log_f N$ and $\log_{2f} N$, respectively. Thus, the height of B is $1/\log_2 f + 1$ ($= \log_f N / \log_{2f} N$) times smaller than that of A. In disk-based indexes, the typical node size varies from 4KB to 64KB. Assuming that the node size is 8KB and nodes are 70% full, f is 716 ($\cong 8192 \times 0.7/8$) for a B+-tree index and about 286 ($\cong 8192 \times 0.7/20$) for a two-dimensional R-tree. Thus, $1/\log_2 f$ is typically around 0.1. On the other hand, the node size is small in main memory indexes [4]. With a node occupying two cache blocks or 128B, f is about 11 for a B+-tree and about 4 for a two-dimensional R-tree. Thus, $1/\log_2 f$ is 0.29 for the B+-tree and 0.5 for the R-tree. In summary, node compression can reduce the height of main memory indexes significantly because the size of nodes is small.

Clustering has been studied extensively in disk-based index structures. In terms of clustering, the B+-tree is optimal in one-dimensional space, but no optimal clustering scheme is known for the multidimensional case. Instead, many heuristic schemes have been studied in various multidimensional index structures [6] [7] [12][13][14]. Our work can be used with most of these clustering schemes.

3. MBR Compression

Here are two desirable properties of the MBR compression scheme that we seek.

Overlap Check Without Decompression: A basic R-tree operation is to check whether each MBR in a node overlaps a given query rectangle. Checking the overlap of two MBRs should be done directly with the compressed MBRs stored in the nodes, without decompressing them. This property enables the basic R-tree operation to be processed with the one-time compression of the query rectangle instead of the decompression of all the compressed MBRs in the encountered nodes.

Simplicity: Compression and decompression should be computationally simple and can be performed only with already cached data. Conventional lossless compression algorithms such as the one used in the GNU gzip program are expensive in terms of both computation and memory access because most of them maintain an entropy-based mapping table and look up the table for compression and decompression [15]. Thus, although they may be useful for disk-resident indexes, they are not adequate for main memory indexes.

3.1 RMBR

An obvious compression scheme is to represent keys relatively within a node [16]. If we represent the coordinates of an MBR relatively to the lower left corner of its parent MBR, the resultant relative coordinates have many leading 0's. By cutting off these leading 0's and recording the number of bits cut off, we can effectively reduce the size of an MBR.

Definition 1. (Relative Representation of MBR or RMBR) Let *P* and *C* be MBRs, that are represented by their lower left and upper right coordinates (*xl*, *yl*, *xh*, *yh*), and let *P* enclose *C*. Then, the relative representation of *C* with respect to *P* has the coordinates relative to the lower left corner of *P*.

$$RMBR_P(C) = (C.xl - P.xl,\ C.yl - P.yl,\ C.xh - P.xl,\ C.yh - P.yl)$$

However, the following simple analysis shows that the RMBR technique can save only about 32 bits per MBR. For simplicity, we assume that the coordinates of MBR are uniformly distributed in their domain and that R-tree nodes of the same height have square-like MBRs roughly of the same size [8]. Without loss of generality, we assume that the domain of *x* coordinates has the unit length and consists of 2^{32} different values equally spaced. Let *f* be the average fanout of leaf nodes, and let *N* be the total number of data objects. Then, there are roughly *N/f* leaf nodes, whose MBRs have the area of *f/N* and the side length of $\sqrt{f/N}$ along each axis. Since there are 2^{32} different values in the unit interval along each axis, there are $2^{32}\sqrt{f/N}$ different values in the interval with the length of $\sqrt{f/N}$. Therefore, we can save 32– $\log_2(2^{32}\sqrt{f/N})$ bits or $\log_2\sqrt{N/f}$ bits for each *x* coordinate value. When *N* is one million and *f* is 11, about 8.2 bits are saved. By multiplying 4, we can save about 32 bits per MBR. Note that the number of saved bits does not depend on the original number of bits as long as the former is smaller than the latter.

We can easily extend this analysis result such that the number of bits saved is parameterized further by the dimensionality. The extended result is $\log_2\sqrt[d]{N/f}$ or

$$(\log_2 N - \log_2 f)/d \qquad (1)$$

The formula (1) increases logarithmically with *N*, decreases logarithmically with *f*, but decreases linearly with *d*. Therefore, the number of saved bits mainly depends on the dimensionality. In one-dimensional space, the relative representation technique can save almost 16 bits for each scalar, but it becomes useless as the dimensionality increases.

3.2 QRMBR

Since we cannot obtain a sufficient compression ratio from the RMBR technique alone, we introduce the additional quantization step. This step cuts off trailing insignificant bits from an RMBR while the RMBR technique cuts off leading non-discriminating bits from an MBR. After defining QRMBR, we show that quantizing an RMBR does not harm the correctness of index search and its small overhead by quantization is paid off by the significant savings in cache misses.

Definition 2. (Quantized Relative Representation of MBR or QRMBR) Let *I* be the reference MBR, and let *l* be the desired quantization level. Then, the corresponding quantized relative representation of an MBR *C* is defined as

$$QRMBR_{I,l}(C) = (\phi_{I.xl,I.xh,l}(C.xl), \phi_{I.yl,I.yh,l}(C.yl), \Phi_{I.xl,I.xh,l}(C.xh), \Phi_{I.yl,I.yh,l}(C.yh))$$

where $\phi_{a,b,l} : R \rightarrow \{0,...,l-1\}$ and $\Phi_{a,b,l} : R \rightarrow \{1,...,l\}$ are

$$\phi_{a,b,l}(r) = \begin{cases} 0 & \text{, if } r \le a \\ l-1 & \text{, if } r \ge b \\ \lfloor l(r-a)/(b-a) \rfloor & \text{, otherwise} \end{cases}$$

$$\Phi_{a,b,l}(r) = \begin{cases} 1 & \text{, if } r \le a \\ l & \text{, if } r \ge b \\ \lceil l(r-a)/(b-a) \rceil & \text{, otherwise} \end{cases}$$

Computational Cost. Lemma 1 says that QRMBR satisfies the first of two desirable properties mentioned at the beginning of this section. Therefore, the computational overhead of QRMBR technique is the cost of compressing the query rectangle into a QRMBR for each visited node. In our implementation, compressing an MBR into a QRMBR consumes at most about 60 instructions, which corresponds to less than 120 ns on a 400 MHz processor because of pipelining. In addition, it incurs no memory access as long as the query MBR and the MBR of the node on immediate access are cached.

Lemma 1. Let *A* and *B* be MBRs. For any MBR *I* and integer *l*, it holds that if $QRMBR_{I,l}(A)$ and $QRMBR_{I,l}(B)$ do not overlap, *A* and *B* also do not overlap.

Proof. See Appendix A. ■

Correctness. Since it is generally not possible to recover the original coordinates of an MBR from its QRMBR, there is the possibility of incorrectly determining the overlap relationship between two MBRs. Lemma 1 guarantees that there is no possibility of saying two actually overlapping MBRs do not overlap. Thus, the QRMBR technique does not miss an object that satisfies a query.

However, there is the possibility of concluding that two actually non-overlapping MBRs overlap. This means that the result of index search may contain false hits that have to be filtered out through a subsequent refinement step. However, this refinement step is needed for most multidimensional index structures because MBRs are typically approximations of objects [9]. Thus, requiring the refinement step itself is not an overhead, but the number of false hits can be. Section 5.3 shows that the number of false hits can be made negligibly small, typically fewer than one, by choosing the quantization level properly.

4. CR-tree

4.1 Algorithms

4.1.1 Searching

The search algorithm is similar to the one used in other R-tree variants. The only difference is that the CR-tree compares a query rectangle with QRMBRs. Instead of recovering MBRs from QRMBRs, the CR-tree transforms the query rectangle into the corresponding QRMBR using the MBR of each node as the reference MBR. Then, it compares two QRMBRs to determine whether they overlap.

Algorithm Search. Given a CR-tree and a query rectangle *Q*, find all index records whose QRMBRs overlap *Q*.

1. Push the root node to the initially empty stack *S*
2. If *S* is empty, stop
3. Pop a node *N* from *S* and set *R* to be $\text{QRMBR}_{N.MBR,1}(Q)$
4. If *N* is not a leaf, check each entry *E* to determine whether *E.QRMBR* overlaps *R*. If so, push *E.ptr* to *S*
5. If *N* is a leaf, check each entry *E* to determine whether *E.QRMBR* overlaps *R*. If so, add *E.ptr* to the result set
6. Repeat from step 2

4.1.2 Insertion

To insert a new object, the CR-tree traverses down from the root choosing the child node that needs the least enlargement to enclose the object MBR. When visiting an internal node to choose one of its children, the object MBR is first transformed into the QRMBR using the reference MBR. Then, the enlargement is calculated between a pair of QRMBRs. When a leaf node is reached, the node MBR is first adjusted such that it encloses the object MBR. Then, an index entry for the object is created in the node. If the node MBR has been adjusted, the QRMBRs in the node are recalculated because their reference MBR has been changed. If the node overflows, it is split and the split propagates up the tree.

Algorithm Insert. Insert a new object *O* whose MBR is *C* into a CR-tree by invoking *ChooseLeaf* and *Install*. The algorithms *SplitNode* and *AdjustTree* can also be invoked if needed. This algorithm is same as that of other R-tree variants.

Algorithm ChooseLeaf. Select a leaf node to insert a new MBR *C*, descending a CR-tree from the root. This algorithm is same as that of other R-tree variants.

Algorithm Install. Install a pair of an MBR *C* and an object pointer *p* in a node *N*.

1. Enlarge *N.MBR* such that it encloses *C*
2. Make an entry of ($\text{QRMBR}_{N.MBR,1}(C)$, *p*) and append it to *N*
3. If *N.MBR* has been enlarged, recalculate all the QRMBRs in *N* by accessing their actual MBRs and invoke **AdjustTree** passing *N*

Algorithm SplitNode. The CR-tree can use the split algorithms used in other R-tree variants including the R-tree and the R*-tree [6][7]. In our experiment, the linear-cost split algorithm of the original R-tree was used. After splitting a node into two, the QRMBRs in the nodes are recalculated according to their MBR.

Algorithm AdjustTree. Ascend from a leaf node *L* up to the root, adjusting MBRs of nodes and propagating node splits as necessary. When a node MBR has been adjusted, recalculate the QRMBRs in the node.

4.1.3 Deletion

Algorithm Delete. Remove index record *E* from a CR-tree. The CR-tree can use any of the deletion algorithms used in the R-tree and the R*-tree. However, the **CondenseTree** algorithm invoked by the **Delete** algorithm needs a slight modification.

Algorithm CondenseTree. Given a leaf node *L* from which an entry has been deleted, eliminate the node if it has too few entries and relocate its entries. Propagate node elimination upward as necessary. Adjust all MBRs of the nodes on the path to the root, making them smaller if possible. When a node's MBR has been adjusted, recalculate the QRMBRs in the node. This last step is what is different from other R-tree variants.

4.1.4 Bulk Loading

Bulk loading into a CR-tree is not different from that into other R-tree variants. As long as QRMBRs are correctly maintained, existing bottom-up loading algorithms can be used directly [17][18].

4.2 Variants and Space Comparison

This paper also considers three variants of the CR-tree: PE (pointer-eliminated) CR-tree, SE (space-efficient) CR-tree, and FF (false-hit free) CR-tree.

The PE CR-tree eliminates most pointers to child nodes from internal nodes as in the CSB+-tree. This extension can widen the CR-tree significantly because the key size of the CR-tree is now small unlike the R-tree. For example, when the size of QRMBR is four bytes, this pointer elimination doubles the fanout of internal nodes. However, it is just a minor improvement in most cases because pointers to data objects stored in leaf nodes can rarely be eliminated. When the average fanout of both internal and leaf nodes is 10, the number of internal nodes is about a ninth of that of leaf nodes. Therefore, the overall increase of fanout is only about 10%. On the other hand, as in the CSB+-tree, node split becomes expensive. The new node created by a split has to be

	Maximum fanout		Node space		Typical index size
	Internal	Leaf	Internal	Leaf	
R-tree	*m*	*m*	*NS*/0.7*m*(0.7*m*-1)	*NS*/0.7*m*	38.15 MB
PE R-tree	1.25*m*	*m*	*NS*/0.7*m*(0.875*m*-1)	*NS*/0.7*m*	35.90 MB
CR-tree	2.5*m*-4	2.5*m*-4	*NS*/(1.75*m*-2.8)(1.75*m*-1.8)	*NS*/(1.75*m*-2.8)	17.68 MB
PE CR-tree	5*m*-5	2.5*m*-4	*NS*/(1.75*m*-2.8)(3.5*m*-2.5)	*NS*/(1.75*m*-2.8)	16.71 MB
SE CR-tree	5*m*-1	2.5*m*-2	*NS*/1.75*m*(3.5*m*-0.7)	*NS*/(1.75*m*-1.4)	14.07 MB
FF CR-tree	2.5*m*-4	*m*	*NS*/0.7m(1.75*m*–2.8)	*NS*/0.7*m*	32.84 MB

Table 2: Space Analysis (*N*: the number of leaf node entries, *S*: the node size in bytes; typical sizes are given when *N*=1,000,000 and *S*=128)

stored consecutively with its siblings, and this often requires allocating a new space and moving the siblings.

The SE CR-tree removes the reference MBR from nodes of the PE CR-tree. This is possible because the reference MBR of a node can be obtained from the matching entry in its parent node. This extension increases the fanout of internal nodes by four and that of leaf nodes by two when the MBR size is 16 bytes and the QRMBR size is 4 bytes. This increase can be larger than the increase obtained in the PE CR-tree when the node size is as small as one or two cache blocks.

While the above two extensions increase the fanout, the third extension to the CR-tree decreases the fanout of leaf nodes. Since the QRMBR technique is a lossy compression scheme, the search result can be a superset of the actual answer for a given query. This can be avoided if we apply the QRMBR technique only to internal nodes and store actual MBRs in leaf nodes. Called the FF CR-tree, this extension is useful when the subsequent refinement step is extremely expensive.

Table 2 shows the space requirements of the various index structures used in this paper, assuming all the nodes are 70% full. We assume that the size of MBR is 16 bytes, the size of QRMBR is 4 bytes, and the size of pointer is 4 bytes. The internal node space is calculated by dividing the leaf space by the average fanout of internal nodes minus one. This analysis shows that the PE CR-tree is not so different from the CR-tree in terms of the space requirement and the PE R-tree is no different from the R-tree.

5. Analysis

Without loss of generality, we assume the data domain of unit hyper-square. For simplicity, we assume that data objects are uniformly distributed in the domain, and the query MBRs are hyper-squares. We further assume that the R-tree nodes of the same height have square-like MBRs roughly of the same size as in other analytical work [8][17].

5.1 Number of Accessed Nodes

Let h denote the height or level of a node assuming that the height of leaf nodes is one. Let M_h denote the number of nodes at the height of h. Then, from the above assumption,

$$M_h = \left\lceil \frac{N}{f^h} \right\rceil .$$

Let a_h denote the average area that a node of height h covers. Then, a_h is $1/M_h$. Using the Minkowski sum technique [17][19], the probability that a node of height h overlaps a given query rectangle is $(\sqrt[d]{s} + \sqrt[d]{a_h})^d$, where s denotes the size of the query rectangle. Then, the number of height-h nodes that overlap the query rectangle is $M_h(\sqrt[d]{s} + \sqrt[d]{a_h})^d$ or

$$\left(1 + \sqrt[d]{\left\lceil \frac{N}{f^h} \right\rceil \cdot s}\right)^d .$$

By summing this equation from the leaf to the root, the total number of node accesses in R-trees is

$$1 + \sum_{h=1}^{\lceil \log_f N \rceil - 1} \left(1 + \sqrt[d]{\left\lceil \frac{N}{f^h} \right\rceil \cdot s}\right)^d . \qquad (2)$$

The CR-tree accesses slightly more nodes than the R-tree because the QRMBR is bigger than the original MBR by the quantization error.

Let l denote the quantization level. Then, each node has l^d quantization cells, and the side length of each cell is $\sqrt[d]{a_h}/l$, where h denotes the height of the node. Since whether to visit a child node is determined by comparing the QRMBR of the query rectangle and the stored QRMBR of the child node, the probability to visit a child node is $(\sqrt[d]{s} + \sqrt[d]{a_h}/l + \sqrt[d]{a_{h-1}} + \sqrt[d]{a_h}/l)^d$. By multiplying by M_h and summing from the leaf to the root, the total number of node accesses in CR-trees is

$$1 + \sum_{h=1}^{\lceil \log_f N \rceil - 1} \left(1 + \sqrt[d]{\left\lceil \frac{N}{f^h} \right\rceil \cdot s} + \sqrt[d]{\left\lceil \frac{N}{f^{h+1}} \right\rceil \cdot s / l}\right)^d . \qquad (3)$$

Figure 3 compares equations (2) and (3) when the cardinality is one million and the query selectivity is 0.01%. Here, we assumed that the pointer size is 4 bytes and that each node is 70% full. The MBR size is 16 bytes in 2D and increases linearly with dimensions. The QRMBR size is a one-fourth of the MBR size. In this figure, the number of node accesses decreases with the node size. The decrease rate is initially large, but it becomes smaller as the node size increases. For all the node sizes and all the three

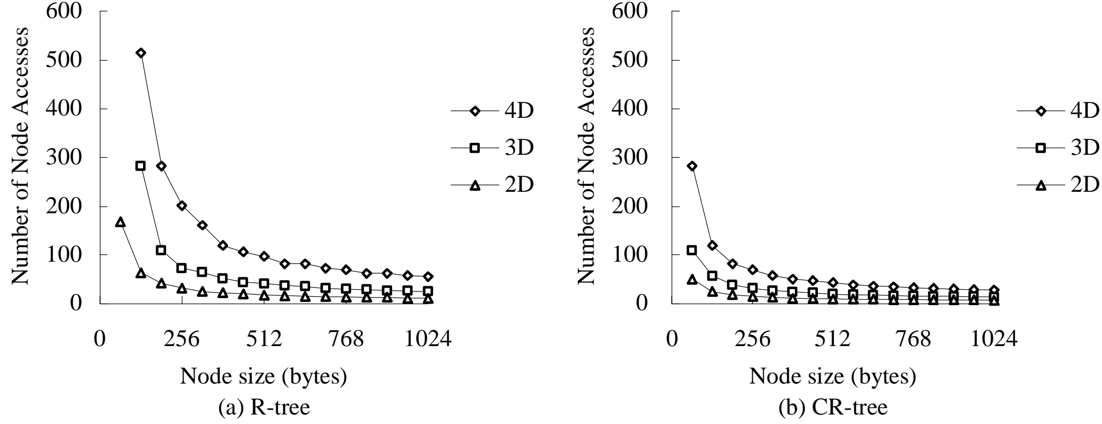

Figure 3: Number of Node Accesses in R-trees and CR-trees (N = 1M, s = 0.01%, MBR: QRMBR = 4:1)

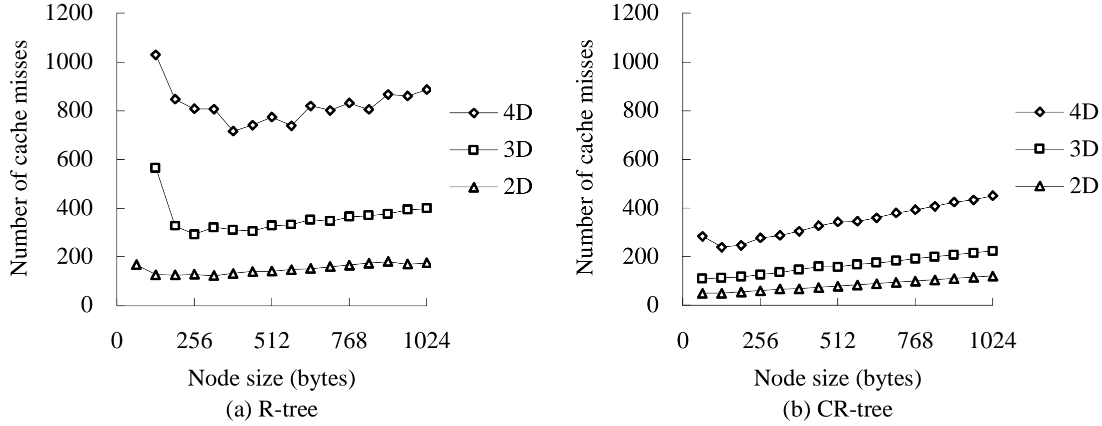

Figure 4: Number of Cache Misses in R-trees and CR-trees (N = 1M, s = 0.01%, MBR:QRMBR = 4:1)

dimensionalities, the CR-tree surpasses the R-tree by more than twice.

5.2 Number of Cache Misses

The number of cache misses can be easily obtained by multiplying equations (2) and (3) by the number of cache misses that one node access incurs. Figure 4 shows the analyzed number of cache misses. It shows that as the node size grows, the number of cache misses approaches quickly to the minimum, and then increases slowly. In terms of cache misses, the CR-tree outperforms the R-tree significantly, by up to 4.3 times. To obtain this figure, the equations (2) and (3) were multiplied by S/64, where S is the node size in bytes and 64 is the L2 cache block size.

Figure 4(a) shows a saw-like pattern that the number of cache misses decreases abruptly at certain node sizes while generally increasing with the node size. Such bumps occur when the height of tree becomes smaller. For example, the 4D R-tree has the height of 7 when the node size is 448 or 512 bytes, but its height becomes 6 when the node size is 576 bytes. In other words, such bumps occur when the gain by the decrease of height surpasses the overhead associated with the increase of node size.

Although the optimal one-dimensional node size in terms of the number of cache misses is shown to be the cache block size in section 2.3, Figure 4 shows that this choice of node size is not optimal in multidimensional cases as discussed in section 2.3. Figure 5 shows the number of cache misses computed changing the query selectivity. The observation on this figure is that the optimal node size increases with the query selectivity in both the R-tree and the CR-tree. Figure 5(a) shows that the optimal node size increases in the order of 128, 192, 320, 640, and 960 bytes as the selectivity increases. Figure 5(b) shows that the optimal node size increases in the order of 64, 128, 192, 256, and 320 bytes as the selectivity increases. Although we do not visualize because of the space limitation, the optimal node size increases in the same way as the cardinality and the dimensionality increase.

5.3 Ratio of False Hits By Quantization

Following the same steps as in section 5.1, each quantization cell of a leaf node has the area of $f/l^d N$ and the side length of $\sqrt[d]{f/l^d N}$ along each axis, and the probability that the QRMBRs

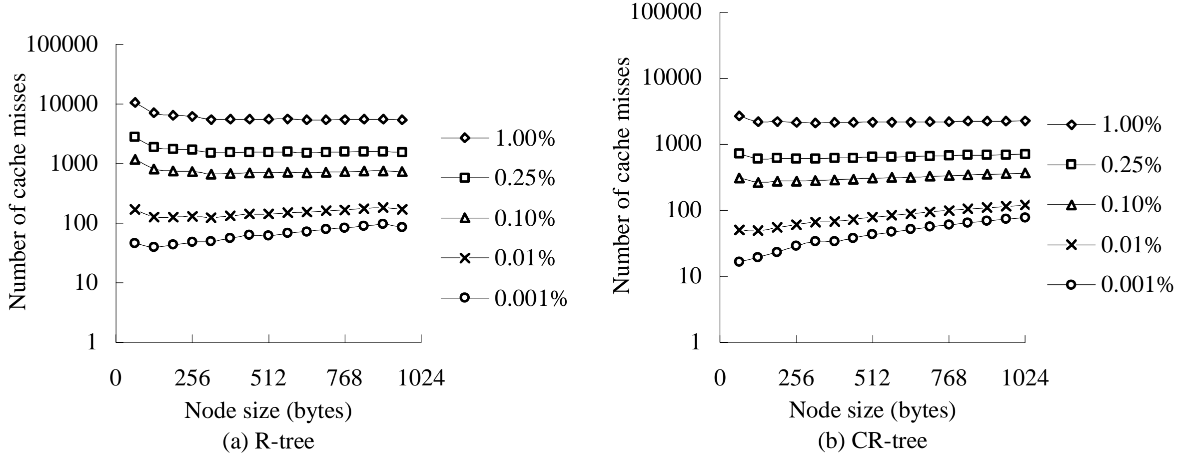

Figure 5: Increase of Optimal Node Size with Selectivity in 2D R-trees and CR-trees

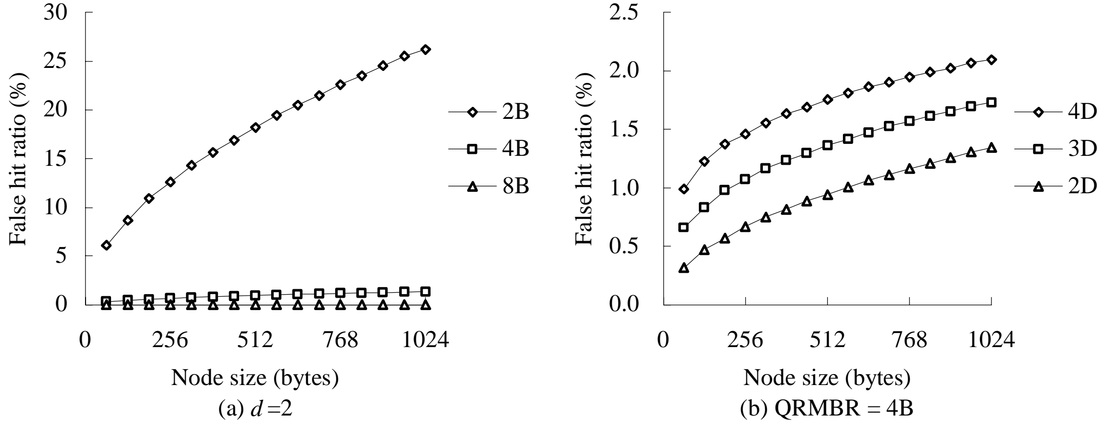

Figure 6: False Hit Ratio by QRMBR Size and Dimensionality (N = 1M, s = 0.01%)

of the query MBR and the object MBR overlap is $\left(\sqrt[d]{s}+\sqrt[d]{a}+2\sqrt[d]{f/l^d N}\right)^d$.

Therefore, the probability that a false hit occurs is $\left(\sqrt[d]{s}+\sqrt[d]{a}+2\sqrt[d]{f/l^d N}\right)^d - \left(\sqrt[d]{s}+\sqrt[d]{a}\right)^d$. Dividing by $\left(\sqrt[d]{s}+\sqrt[d]{a}\right)^d$, the ratio of false hits incurred by quantization to actual answers is

$$\left(1+2\sqrt[d]{f/l^d N}\,/\left(\sqrt[d]{s}+\sqrt[d]{a}\right)\right)^d-1\,. \quad (4)$$

Figure 6 shows the ratio when the cardinality is one million and the query selectivity is 0.01%. Here, we assume that the pointer size is 4 bytes and that each node is 70% full. Figure 6(a) shows the false hit ratio in the 2D CR-tree for three different QRMBR sizes: 2 bytes, 4 bytes, and 8 bytes, and Figure 6(b) shows the false hit ratio for three different dimensionality. The false hit ratio increases with both the node size and the dimensionality. Using QRMBRs of 4 bytes incurs around one false hit in this configuration, but it saves tens or hundreds of cache misses as shown in Figure 4.

6. Experimental Evaluation

To assess the merit of the proposed CR-tree and its variants, we conducted a series of experiments on a SUN UltraSPARC platform (400MHz CPU with 8MB L2 cache) running Solaris 2.7.

We implemented six index structures on 2D: the ordinary R-tree, the PE R-tree, the CR-tree, the PE CR-tree, the SE CR-tree, and the FF CR-tree. We also implemented a bulk-loading algorithm [17]. We changed the size of nodes from 64 bytes to 1024 bytes for the implemented index structures. We used 16-byte MBRs and changed the size of QRMBRs from 2 bytes to 8 bytes. If not specified, the default size of QRMBRs is 4 bytes, and the nodes are 70% full.

We generated two synthetic data sets consisting of one million small rectangles located in the unit square. One is uniformly distributed in the unit square and the other has the Gaussian distribution around the center point (0.5, 0.5) with the standard deviation of 0.25. We set the average side length of rectangles to be 0.001.

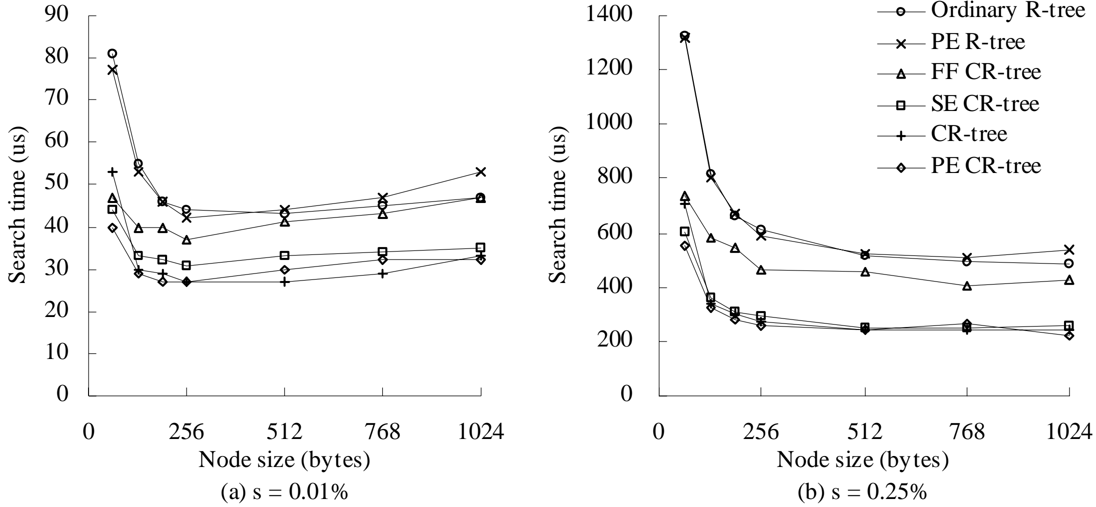

Figure 7: Search Performance of Bulk-loaded 2D Indexes with Uniform Data Set

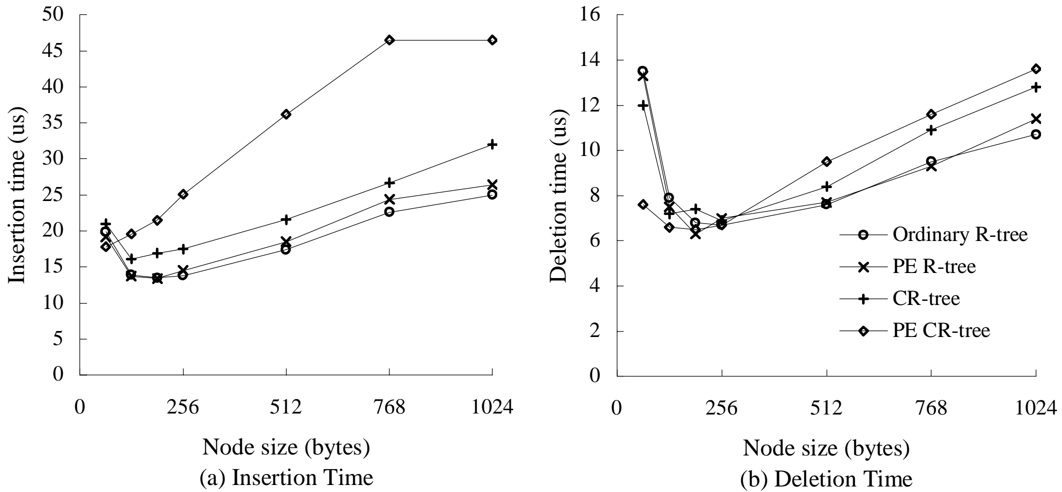

Figure 8: Update Performance on Bulk-loaded 2D Indexes with Uniform Data Set

6.1 Search Performance

In the first experiment, we compare the search performance of various indexes in terms of the wall-clock time spent processing a two-dimensional region query. We generated 10,000 different query rectangles of the same size, whose center points are uniformly distributed. We changed the size of query rectangles from 0.01% of the data space to 1%. Since the data space is the unit square, the query selectivity is roughly same as the size of a query rectangle.

Figure 7 shows the elapsed time spent searching various indexes bulk-loaded with the uniform data set such that each node is 70% full. The observations on this figure are:

- As the node size grows, the search time approaches quickly to the minimum, and then increases slowly. The minimum shifts to the right as the selectivity increases. This trend holds for all the six trees, and it coincides with the analytical results presented in section 5.1.
- The CR-tree, the PE CR-tree, and the SE CR-tree form the fastest group. The R-tree and the PE R-tree form the slowest group. The FF CR-tree lies between the two groups.
- Although the SE CR-tree is wider than both the CR-tree and the PE CR-tree, it performs worse. This is because the SE CR-tree calculates the reference MBR of a node from the matching entry in its parent node. In our implementation, this calculation involves about 40 instructions and 16 bytes of memory write.

We conducted the same experiment for the skewed data set. We could not find any noticeable difference from Figure 7. In other words, all the six trees are robust to the skew for any node size.

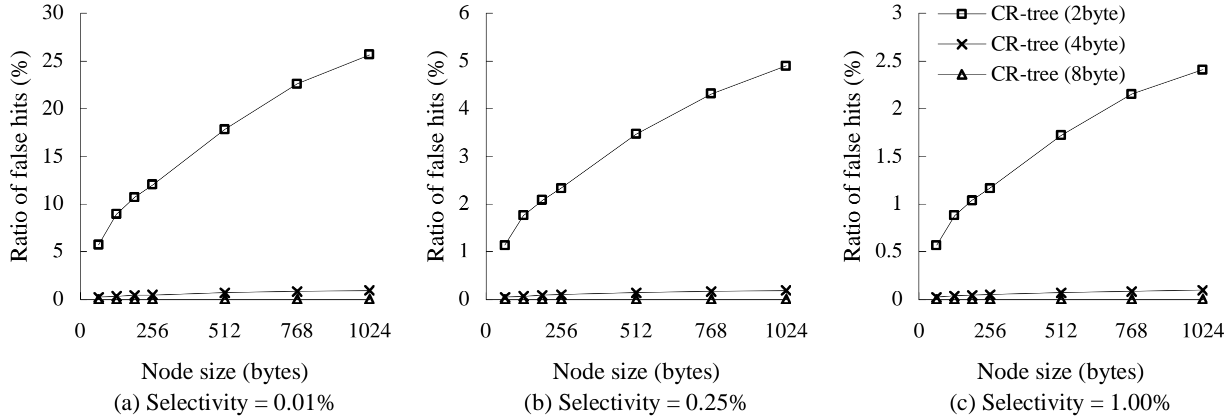

Figure 9: Ratio of False Hits Incurred by Quantization

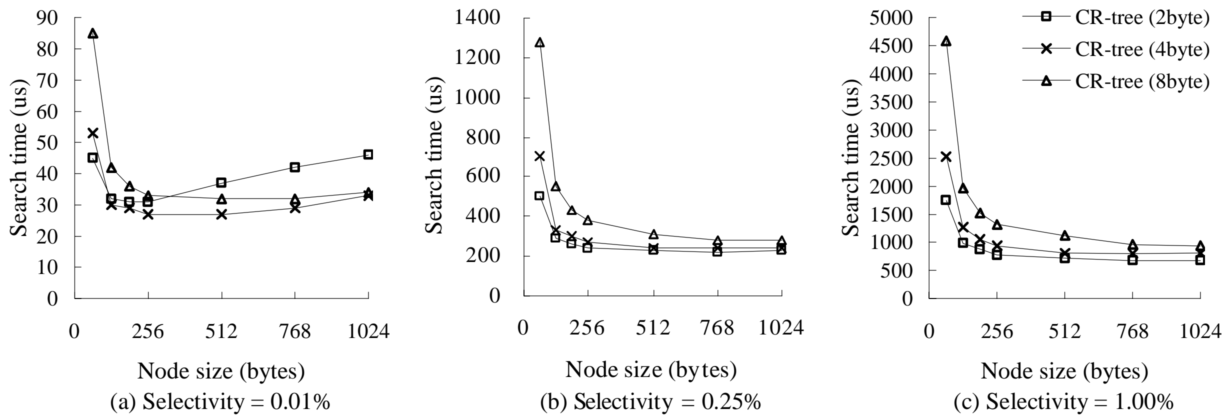

Figure 10: Search Time with Varying Quantization Levels

6.2 Update Performance

To measure the update performance, we inserted 100,000 objects into indexes bulk-loaded with the one million uniform data set, then removed 100,000 randomly selected objects from the indexes.

Figure 8(a) and (b) show the measured elapsed time per insertion and deletion, respectively. For a given node size, the CR-tree consumes about 15% more time than the R-tree for insertion. However, when the fanout is same (for example, the CR-tree with the node size of 256 bytes and the R-tree with the node size of 640 bytes), the CR-tree performs similarly to or better than the R-tree. This can be explained in the following way.

When descending a tree for insertion, the child node that needs to be enlarged least is selected. Since the enlargement calculation consumes about 30 instructions in our implementation, it becomes more expensive than the cache miss in the CR-tree and its variants. Since a single cache block contains about 5.6 QRMBRs in the CR-tree, the enlargement calculation cost is about 168 instructions per cache block, but a cache miss consumes about 80~100 processor cycles on 400MHz UltraSPARC II. On the other hand, since insertion accesses only one node for each height, the number of accessed nodes decreases logarithmically with the fanout, but the number of enlargement calculations for each node increases linearly with the fanout. Thus, the total number of enlargement calculations increases with the fanout.

The PE R-tree performs slightly worse than the R-tree because it increases the fanout by less than 25%. Since the fanout of the CR-tree is about 150% larger than that of the R-tree, it performs worse than the R-tree for a given node size. Since the fanout of the PE CR-tree is about 400% larger than that of the R-tree, it performs significantly worse than the R-tree for a given node size. On the other hand, when the fanout is same, the ranking of the CR-tree is determined by the saving in cache misses and the overhead of updating QRMBRs when the node MBR grows or shrinks.

Figure 8(b) shows that the rankings for deletion are slightly different from those for insertion. Deletion is a combination of highly selective search and node update. As you can expect from Figure 7, the CR-tree performs similarly to the R-tree as the selectivity decreases. On the other hand, node update becomes more expensive as the node size increases because the cost of updating QRMBRs increases. Therefore, the CR-tree outperforms the R-tree when the node size is small, but they cross over as the node size increases.

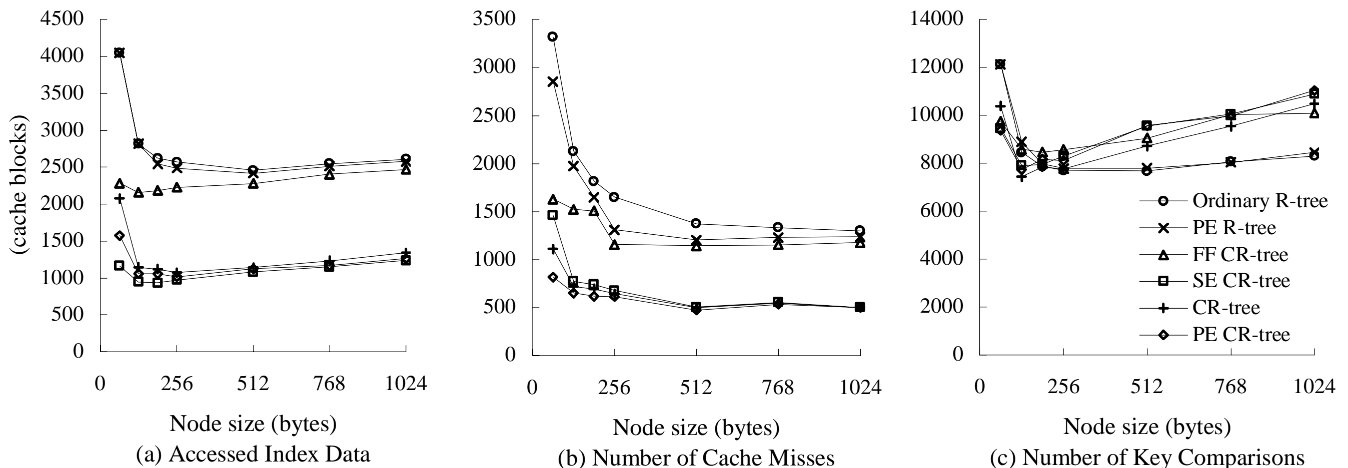

Figure 11: Breakdown of Search Performance (s = 0.25%)

6.3 Impact of Quantization Levels

To assess the impact of quantization levels, we measured the ratio of false hits incurred by quantization and the search time for three different quantization levels, 2^4, 2^8, and 2^{16}. These correspond to QRMBRs of 2 bytes, 4 bytes, and 8 bytes, respectively. In this experiment, we used the trees bulk-loaded with the 1M uniform data set.

Figure 9 shows the ratio of false hits measured varying the quantization level. In section 5.3, we have shown that the ratio of false hits can be estimated by $\left(1+2\sqrt{f/l^2N}\,/\left(\sqrt{s}+\sqrt{a}\right)\right)^2-1$. This ratio increases with the fanout or the node size, and decreases with the increasing quantization level and selectivity. Figure 9 is consistent with the analytical result. With the 16-bit quantization, the result of the CR-tree search is almost the same as that of the R-tree search. With the 8-bit quantization, the CR-tree search result contains at most 1% more objects than the R-tree result. With the 4-bit quantization, the ratio of the false hits increases steadily with the node size, up to 26% when the node size is 1024 bytes and the selectivity is 0.01%. When the selectivity is high, the graph shows a similar slope with respect to the selectivity but the ratio of the false hits is contained within a few percents. So the 4-bit quantization becomes useful as the selectivity increases.

Figure 10 shows the effect of the quantization on the search time. The time for filtering out the false hits is not counted. The figure shows that the 8-bit quantization performs the best when the selectivity is 0.01%. The 4-bit quantization with 0.01% selectivity performs well when the node size is small but becomes the worst as the node size grows. However, the 4-bit quantization performs the best regardless of the node size when the selectivity is high. This is because the number of false hits becomes relatively insignificant as the node size grows.

6.4 Breakdown of Search Performance

To better understand the search performance of the indexes used in our experiment, we measured the amount of accessed index data, the number of L2 cache misses, and the number of key comparisons for the experiment reported in Figure 7.

Figure 11(a) shows the amount of accessed index data, which is the number of L2 cache misses when no index data is cached initially or the worst-case cache misses. In terms of the worst-case cache misses, the six trees are clearly ranked by their fanout or in the order of the SE CR-tree, the PE CR-tree, the CR-tree, the FF CR-tree, the PE R-tree, and the R-tree, from the best to the worst. The first three form one group, and the last two form another group as in Figure 7. This result coincides with Figure 4.

Figure 11(b) shows the measured number of L2 cache misses using the *Perfmon* tool [20]. The UltraSPARC processors provide two registers for measuring processor events. We used the *Perfmon* tool to make these registers count L2 cache misses and to read the values stored in them. The number of L2 cache misses is slightly different from the amount of accessed index data because of cache hits and missing instructions. Instruction cache misses explains why the number of measured cache misses can be larger than that of the worst-case cache misses in Figure 11(a) when both the node size and the selectivity are small.

Another observation on Figure 11(b) is that the cache hit ratio increases with the node size. This has to do with the typical cache replacement policy based on the circular mapping of memory blocks to cache blocks. Namely, the memory block with the address A is cached into the cache block whose address is determined by the cache size modulo of A. With this policy, a node consuming multiple memory blocks is placed consecutively in the cache. As the node size increases, the probability that the concurrently needed memory blocks are mapped to the conflicting location of the cache decreases.

Figure 11(c) shows that the QRMBR technique increases the number of key comparisons slightly. Since the overlap test between two MBRs consumes less than 10 instructions on average in our implementation, saving an L2 cache miss is worth saving at least 10 overlap tests. The R-tree and the PE R-tree have similar fanouts and form one group. The PE CR-tree and the SE CR-tree also have similar fanouts and form another group.

7. Conclusion

There has been much research on multidimensional indexes. This paper addressed the problem of optimizing the cache behavior of multidimensional indexes for use in the main memory database environment. To pack more entries in the node whose size is given in multiples of cache blocks, we have proposed an efficient MBR compression scheme called the quantized relative representation of MBR or QRMBR which represents the coordinates of child nodes relatively to the MBR of the parent node and quantizes the resultant relative MBR using a fixed number of bits. The CR-tree based on QRMBR effectively increases the fanout of the R-tree and decreases the index size for the improved cache behavior.

Our extensive experimental study combined with analytical one shows that the 2D CR-tree and its three variants outperform the ordinary R-tree up to 2.5 times in the search time and use about 60% less memory space. To see the practical impact of the CR-tree, we are currently integrating the CR-tree into P*TIME, a prototype transact in memory engine under development.

8. References

[1] P. Boncz, S. Manegold, and M. Kersten, "Database Architecture Optimized for the New Bottleneck: Memory Access", *Proceedings of VLDB Conference*, 1999, pp. 54-65.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, D. A. Wood, "DBMSs on a Modern Processor: Where Does Time Go?", *Proceedings of VLDB Conference*, 1999, pp. 266-277.

[3] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd Edition, Morgan Kaufmann, 1996.

[4] J. Rao, K. A. Ross, "Cache Conscious Indexing for Decision-Support in Main Memory", *Proceedings of VLDB Conference*, 1999, pp. 78-89.

[5] J. Rao, K. A. Ross, "Making B+-trees Cache Conscious in Main Memory", *Proceedings of ACM SIGMOD Conference*, 2000, pp. 475-486.

[6] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching", *Proceedings of ACM SIGMOD Conference*, 1984, pp. 47-57.

[7] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles", *Proceedings of ACM SIGMOD Conference*, 1990, pp. 322-331.

[8] C. Faloutsos and I. Kamel, "Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension", *Proceedings of ACM PODS Symposium*, 1994, pp. 4–13.

[9] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger, "Multi-Step Processing of Spatial Joins", *Proceedings of ACM SIGMOD Conference*, 1994, pp. 197-208.

[10] Sun Microsystems, *UltarSPARC™ User's Manual*, 1997.

[11] J. M. Hellerstein, "Indexing Research: Forest or Trees?", *Proceedings of ACM SIGMOD Conference*, 2000, pp. 574, Panel.

[12] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-tree: A Dynamic Index for Multi-dimensional Objects", *Proceedings of VLDB Conference*, 1987, pp. 507-518.

[13] I. Kamel and C. Faloutsos, "Hilbert R-tree: An Improved R-tree Using Fractals", *Proceedings of VLDB Conference*, 1994, pp. 500-509.

[14] V. Gaede and O. Günther, "Multidimensional Access Methods", *Computing Surveys*, 30(2), 1998, pp. 170-231.

[15] A. Bookstein, S. T. Klein, T. Raita, "Is Huffman Coding Dead?", *Proceedings of ACM SIGIR Conference*, 1993, pp. 80-87.

[16] J. Goldstein, R. Ramakrishnan, and U. Shaft, "Compressing Relations and Indexes", *Proceedings of IEEE Conference on Data Engineering*, 1998, pp. 370-379.

[17] I. Kamel and C. Faloutsos, "On Packing R-trees", *Proceedings of ACM CIKM Conference*, 1993, pp. 490-499.

[18] S. T. Leutenegger, J. M. Edgington, M. A. Lopez, "STR: A Simple and Efficient Algorithm for R-Tree Packing", *Proceedings of IEEE Conference on Data Engineering*, 1997, pp. 497-506.

[19] S. Berchtold, C. Böhm, H.-P. Kriegel, "The Pyramid-Tree: Breaking the Curse of Dimensionality", *Proceedings of ACM SIGMOD Conference*, 1998, pp. 142-153.

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

Appendix A. Proof of Lemma 1.

We prove the contrapositive that if A and B overlap, QRMBR$_{I,l}(A)$ and QRMBR$_{I,l}(B)$ overlap. By definition, two rectangles overlap if and only if they share at least one point. Thus, A and B share at least one point. Let (x, y) denote this point. Then, the following holds.

$$A.xl \le x \le A.xh, \quad A.yl \le y \le A.yh$$
$$B.xl \le x \le B.xh, \quad B.yl \le y \le B.yh$$

For simplicity, we omit the subscripts a, b, and l from the quantization functions ϕ and Φ. Since, ϕ and Φ are monotonically non-decreasing functions and $\phi(r) \le \Phi(r)$ for any $r \in R$,

$$\phi(A.xl) \le \phi(x) \le \Phi(x) \le \Phi(A.xh),$$
$$\phi(A.yl) \le \phi(y) \le \Phi(y) \le \Phi(A.yh),$$
$$\phi(B.xl) \le \phi(x) \le \Phi(x) \le \Phi(B.xh),$$
$$\phi(B.yl) \le \phi(y) \le \Phi(y) \le \Phi(B.yh)$$

Therefore, QRMBR$_{I,l}(A)$ and QRMBR$_{I,l}(B)$ share at least the point $(\phi(x), \phi(y))$. Thus, they overlap and this completes the proof.■