

# CSR<sup>+</sup>-tree: Cache-conscious Indexing for High-dimensional Similarity Search

Junfeng Dong  
Microsoft Corporation  
One Microsoft Way  
Redmond, WA, USA, 98052-6399  
[junfeng.dong@microsoft.com](mailto:junfeng.dong@microsoft.com)

Xiaohui Yu  
School of Information Technology  
York University  
Toronto, ON, Canada, M3J 1P3  
[xhyu@yorku.ca](mailto:xhyu@yorku.ca)

## Abstract

*In this paper, we propose a novel index structure, the CSR<sup>+</sup>-tree, to support efficient high-dimensional similarity search in main memory. We introduce Quantized Bounding Spheres (QBSs) that approximate Bounding Spheres (BSs) or data points. We analyze the respective pros and cons of both QBSs and the previously proposed Quantized Bounding Rectangles (QBRs), and take the best of both worlds by carefully incorporating both of them into the CSR<sup>+</sup>-tree. We further propose a novel distance computation scheme that eliminates the need for decompressing QBSs or QBRs, which results in significant cost savings. We present an extensive experimental evaluation and analysis of the CSR<sup>+</sup>-tree, and compare its performance against that of other representative indexes in the literature. Our results show that the CSR<sup>+</sup>-tree consistently outperforms other index structures.*

## 1 Introduction

Recent years have seen an ever growing need for supporting high-dimensional similarity queries in many areas such as geography, mechanical CAD, and medicine. A basic operation in similarity search is the  $k$ -nearest neighbors (KNN) query, which searches for the  $k$  objects from the data set that are closest to a given query point. A number of index structures have been proposed for this purpose, such as R-tree [13], R\*-tree [1], SS-tree [22], SR-tree [16], X-tree [4], V-file [21], IQ-tree [2], A-tree [20]. These index structures have largely been studied in the context of disk-based systems, where it is assumed that the databases are too large to fit into the main memory, and therefore a major concern is to reduce the disk I/O. However, as main memory sizes of computer systems become larger, more and more data sets of interest can be held in main memory. It is now feasible to hold many of the index structures in main memory. Consequently, the traditional I/O bottleneck of disk accesses disappears.

Shifting to the main memory data processing paradigm, however, brings new issues and challenges. It is of critical importance to minimize the L2 cache misses, because cache misses incur a substantial penalty as the corresponding cache block must be fetched from the (much slower) main memory. Another source of negative impact on the query performance is the TLB misses. If a logical address is

not found in the TLB, a TLB miss occurs, which introduces a significant penalty (around 100 cycles on an UltraSparc CPU). Besides cache misses and TLB misses, distance computations also contribute significantly to the overall cost of main memory KNN search [5, 9, 6].

A number of index structures have been proposed for main memory query processing [19, 17, 8], built upon the idea of packing the index into pages that can fit in the cache line, the size of which usually ranges from 32 to 128 bytes. However, these indexes are designed for single or low-dimensional data and are not suitable for high-dimensional data. This is because the size of even a single high-dimensional data point can be greater than that of the cache line. The  $\Delta$ -tree [9] and  $\Delta^+$ -tree [6] proposed by Cui et al. are targeted at high-dimensional similarity search. They effectively reduce the sizes of the data points through dimensionality reduction (PCA), aiming at reducing cache misses and distance computations. However, our experiments reveal that for some high-dimensional data sets, there may exist severe overlapping between the bounding spheres enclosing the clusters of points, resulting in high search costs.

We propose a novel index structure, the Cache-conscious SR<sup>+</sup>-tree (CSR<sup>+</sup>-tree), to support efficient KNN search in main memory. We introduce Quantized Bounding Spheres (QBSs) that approximate bounding spheres (BSs) or data points through quantization. By reducing the representation sizes of BSs, more entries can be packed into a fixed-size index node, so the fan-out of the CSR<sup>+</sup>-tree is increased, which helps reduce the number of index nodes accessed during search. In the index structure, we also utilize Quantized Bounding Rectangles (QBRs). We provide an analysis of the respective pros and cons of QBSs and QBRs, and show how we can use them at different levels of the tree to benefit from the best of both worlds.

Besides increasing the fan-out, keeping the distance computation time small is also very important in high-dimensional spaces. Normally decompression is required before the distance computation in all quantization techniques proposed previously. We propose a new distance computation algorithm without decompression to speed up the KNN searches. Rather than decompressing QBSs so that the coordinates of the decompressed QBSs are in the same coordinate system as the query point, for each index node of the CSR<sup>+</sup>-tree, the algorithm transforms the query point such that the coordinates of the transformed query point are

in the same coordinate system as the QBSs contained in the node. To compute the distance between the query point and a QBS in the node, a weighted distance function (described in Section 4) is then applied to the new query point and the QBS.

Our contributions can be summarized as follows:

- We provide a thorough analysis of the major factors affecting the performance of in-memory high-dimensional similarity search.
- We analyze the advantages and disadvantages of QBSs and QBRs, which provides the insight for us to propose the CSR<sup>+</sup>-tree. Through the judicious use of both QBSs and QBRs, we significantly reduce the fan-out of the index structure and thus the time for index access.
- We propose a distance computation algorithm that does not require decompression of QBSs. Since decompression is an expensive part of distance computation, the algorithm results in large cost savings.
- We conducted extensive experiments on two real datasets of different characteristics to evaluate the proposed index structure. We also performed a thorough comparative study of our approach with existing index structures.

The rest of the paper is organized as follows. Section 2 provides a brief review of related work. Section 3 describes the motivation for the CSR<sup>+</sup>-tree, and introduces the notion of QBS, while Section 4 presents the structure of the CSR<sup>+</sup>-tree, and the insertion, search, and distance computation algorithms. Section 5 shows the experimental results, and Section 6 concludes this paper.

## 2 Related Work

Many index structures, including both disk-based indexes and main memory indexes, have been designed specifically for high-dimensional data sets.

The R-tree [13], the R\*-tree [1] and the X-tree [4] use bounding rectangles as page regions to group points/objects, while the SS-tree [22] employs bounding spheres. The SR-tree [16] uses the intersection solid between a bounding rectangle and a bounding sphere as the page region. The A-tree [20] is a hierarchical index structure, derived from the SR-tree and the VA-file [21]. The design of the A-tree centers around the notion of virtual bounding rectangles (VBRs) that approximate MBRs or vectors.

Kim et al. proposed a cache-conscious version of the R-tree called the CR-tree [17], which is specially designed for use in main memory databases. To pack more entries into a node, the CR-tree employs exactly the same quantization technique as in the A-tree to compress MBRs. Since the partition strategy of the CR-tree is the same as that of the R-tree, the CR-tree is not scalable with respect to increasing dimensionality [9].

Cui et al. presented the  $\Delta$ -tree [9], a multi-level main memory index structure that employs a dimensionality reduction technique, Principal Component Analysis (PCA)

$d$	Dimensionality of the data space
$B$	Number of bytes per dimension for a descriptor
$S$	Size of an index node in bytes
$c$	Block size in bytes for L2 cache
$N$	Number of index nodes accessed during search

**Table 1. Notations**

[15], in its indexing scheme. The  $\Delta^+$ -tree [6] improves upon the  $\Delta$ -tree by first performing a global clustering of the data points and then partitioning the clusters into small regions. Their study showed that the  $\Delta^+$ -tree outperforms other well-known schemes in main memory. However, both trees suffer the same problem as the SS-tree does, i.e., using BSs of the clusters can cause significant overlapping, especially for clusters in high dimensional spaces.

## 3 Motivation for the CSR<sup>+</sup>-tree

### 3.1 Problem Formulation

In main memory, the search time of hierarchical index structures consists mainly of the distance computation time, the time for cache misses, and the time for TLB misses. For simplicity, we assume that there is no concurrency among distance computation, cache access time, and TLB processing time. We also omit the time for instruction cache misses because the number of instruction misses mostly depends on the compiler, which is beyond the user’s control. Thus, the total cost for an index search can be approximated as:

$$T_{search} = T_{dist} + T_{cache} + T_{TLB}$$

where  $T_{dist}$  is the distance computation time,  $T_{cache}$  is the time for data cache misses, and  $T_{TLB}$  is the time for TLB misses.

In general, a hierarchical index consists of leaf nodes and non-leaf nodes. A leaf node contains a set of data points, and those data points are ideally spatially “near” each other. A non-leaf node contains a number of descriptors of child nodes, each of which includes (quantized) rectangles and/or (quantized) spheres. Some notations are defined in Table 1. Note that  $B$ ’s value differs depending on whether a descriptor is a sphere or a rectangle or both, and also depending on the degree of quantization chosen. A hyper-sphere is specified by its center (a vector of length  $d$ ) and its radius (i.e.,  $d + 1$  values in all). A hyper-rectangle is specified by two diagonally opposite corners (e.g., the length  $d$  vectors of the corners nearest to and farthest from a specified corner of the data space), requiring  $2d$  values in all. If, for example, each real value is represented by its high-order byte, then spheres, rectangles, and sphere/rectangle pairs lead to  $B$  values of 1, 2, and 3 bytes per dimension, respectively. In the notation of Table 1, the distance computation time can be estimated as:  $T_{dist} = N \cdot \frac{S}{d \cdot B} \cdot t_{dist}$ , where  $t_{dist}$  is the time for computing the distance between the query point and a node entry. In the worst case, one cache miss occurs for loading each cache block. So, the time for cache misses can be expressed as:  $T_{cache} = N \cdot \frac{S}{c} \cdot t_{cache}$ , where  $t_{cache}$  is the time for one cache miss. For simplicity, we assume that no logical addresses of index nodes are cached in the TLB initially. The maximum time for TLB misses can be expressed for the worst case as:  $T_{TLB} = N \cdot t_{TLB}$ , where  $t_{TLB}$  is the

time for one TLB miss. Hence, the total search time is:

$$T_{search} = N \cdot \frac{S}{d \cdot B} \cdot t_{dist} + N \cdot \frac{S}{c} \cdot t_{cache} + N \cdot t_{TLB}$$

To minimize the search time, our goals are to keep small: (1) the total number of entries read,  $N \cdot \frac{S}{d \cdot B}$ , (2) the total number of cache misses,  $N \cdot \frac{S}{c}$ , (3) the total number of index nodes accessed,  $N$ , and (4)  $t_{dist}$ , the unit distance computation time for computing the distance between the query point and a node entry. The fan-out of an index structure is given by:

$$N_{fan-out} = \begin{cases} \left\lfloor \frac{S}{d \cdot b_{leaf}} \right\rfloor & \text{leaf node} \\ \left\lfloor \frac{S}{d \cdot b_{non-leaf} + S_{ptr}} \right\rfloor & \text{non-leaf node} \end{cases}$$

where  $b_{leaf}$  is the number of bytes per dimension for a leaf node entry,  $b_{non-leaf}$  is the number of bytes per dimension for a non-leaf node entry, and  $S_{ptr}$  is the size of a pointer in bytes. Clearly, the fan-out decreases as dimensionality increases. Observe that, by increasing the fan-out of the index structure without increasing the node size, the number of nodes accessed during a search can be reduced. Alternatively, if the fan-out is kept unchanged, by decreasing both  $S$  and either  $B$  or  $d$ , so that  $N$  is kept unchanged, the search time can also be reduced. It is clear that, by reducing either  $d$  or  $B$ , the fan-out will increase. To reduce  $d$ , we can employ dimensionality reduction. To reduce  $B$ , we can use quantization.

### 3.2 BS+BR vs. BR

Consider the following two index structures,  $A_1$  and  $A_2$ .  $A_1$  has both a bounding sphere (BS) and a bounding rectangle (BR) in a node entry (e.g., the SR-tree), and  $A_2$  has only a BS in a node entry (e.g., the SS-tree). Considering the space required to store a BR and a BS, the entry size in  $A_1$  will be three times larger than that in  $A_2$ . The resulting reduced fan-out in  $A_1$  may cause more nodes to be read on queries and may reduce the query performance. Our experiments show that the number of *internal* node reads of  $A_2$  is less than that of  $A_1$ . This implies that using BSs at the internal node levels may be better than using the intersections of BSs and BRs. Our experiments show that as the dimensionality grows, the gap between using BSs alone and using BR and BS intersections at the internal node levels enlarges rather quickly.

On the other hand, a BS has in general a much larger volume than a BR. BSs thus tend to overlap. This causes queries to pursue multiple paths from the root to potentially relevant leaves. Our experience shows that (1) at the leaf-level, BRs have much smaller volumes than the corresponding BSs on average, and (2) the leaf-level BRs have diameters as short as those of the BSs. This is also evidenced by the experiments by Katayama *et al.* [16], where they showed that the number of leaf node reads of the SR-tree is less than that of the SS-tree. This implies that using BRs at the leaf-level may provide as strong a pruning ability as using the intersections of BRs and BSs at the leaf-level.

Based on the above discussion, we propose using BSs at the internal node levels and using BRs at the leaf-level in the  $CSR^+$ -tree. However, instead of using exact bounding spheres and bounding rectangles, we employ Quantized Bounding Spheres (QBSs) and Quantized Bounding Rectangles (QBRs), which are described in the next sub-section. (Our QBRs are conceptually similar to the VBRs of the A-tree [20].)

### 3.3 Quantized Bounding Sphere

To better utilize the L2 cache, the basic strategy is to pack more entries into a fixed-size node. As the analysis in Section 3 suggests, quantization and dimensionality reduction can be used to reduce the representation sizes of MBRs and BSs. We focus on quantization; dimensionality reduction can be an optional preprocessing step. We can always apply dimensionality reduction to the data set first, and then build an index structure over the reduced data set.

A QBS (Figure 1) approximates a bounding sphere, or a data point (which can be represented by a bounding sphere of radius zero). Assuming that we have a bounding rectangle  $R$  in a  $d$ -dimensional space, a number of bits  $b_c$  is assigned to each dimension  $i$  (for instance, 8 bits), and  $R$  is equally divided into  $2^{b_c}$  slices along each dimension  $i$ . Then,  $R$  can be partitioned into  $2^{d \cdot b_c}$  cells, each of which has a center. Each center can be represented by a unique bit-string with a length of  $d \cdot b_c$  bits. Consider a bounding sphere  $S$  whose centroid falls into a cell in  $R$ . The QBS of  $S$  can be obtained by shifting the centroid of  $S$  to the center of the cell and recalculating the radius so that the QBS will include all points associated with  $S$ . Because the centroid of the QBS can be quantized using a bit-string of length  $d \cdot b_c$  bits with respect to the bounding rectangle  $R$ , the QBS can be represented rather compactly. The fan-out of the  $CSR^+$ -tree is bigger than that of other index trees such as the SS-tree and the SR-tree, which leads to fast search. Now, we give the formal definition of the QBS. In a  $d$ -dimensional space, let  $R = (\vec{a}, \vec{a}')$  be a bounding rectangle, where  $\vec{a}$  and  $\vec{a}'$  are dimension  $d$  vectors and represent diagonally opposite corner points of a bounding rectangle. Let  $S = (\vec{c}, r)$  be a bounding sphere whose centroid is contained inside  $R$ , where  $\vec{c}$  is a dimension  $d$  vector and represents the center point of  $S$ . Let  $b_c (\geq 1)$  be the number of bits used to represent the coordinate of a point in each dimension.  $QBS(S, R) = (\vec{c}', r')$ , the QBS of  $S$  with respect to  $R$ , is defined as follows:

$$c'_i = a_i + \frac{(a'_i - a_i) \cdot (v_i + 0.5)}{2^{b_c}} \quad (1)$$

$$r' = \max \{dist(\vec{c}', \vec{p}) | \forall \vec{p} \in S\}$$

where

$$v_i = \begin{cases} \lfloor \frac{c_i - a_i}{a'_i - a_i} \cdot 2^{b_c} \rfloor & a_i \leq c_i < a'_i \\ 2^{b_c} - 1 & c_i = a'_i \end{cases} \quad (2)$$

Intuitively,  $\vec{c}'$  is the center of the quantized bounding sphere, and its radius,  $r'$ , is set just large enough so that the sphere encloses all points  $\vec{p}$  in  $S$ . Then  $\forall i, 0 \leq i < d, v_i \in [0, 2^{b_c} - 1]$ , and the QBS( $S, R$ ), the QBS of  $S$  with respect to  $R$ , can then be represented by  $(\vec{v}, r')$  where  $\vec{v}$  is a bit vector of length  $d \cdot b_c$  bits, and  $r'$  is represented in 32 or 64 bits.

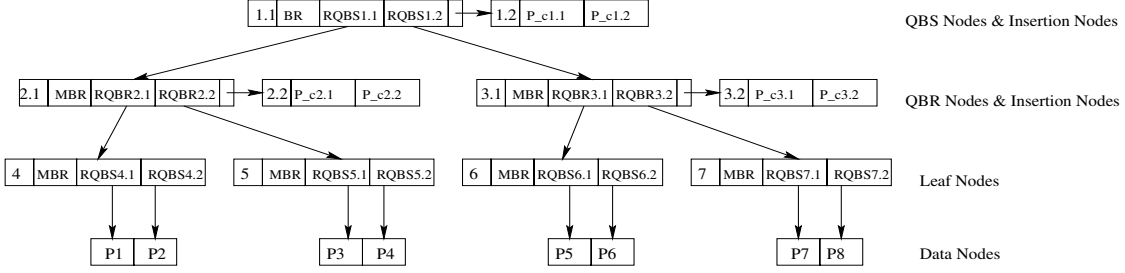


Figure 2. Structure of the CSR<sup>+</sup>-tree

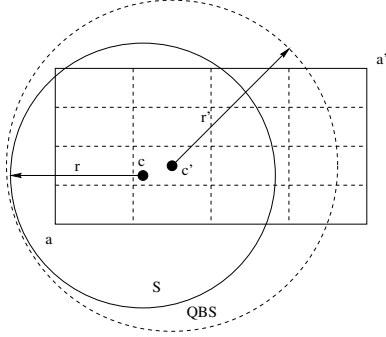


Figure 1. Quantized Bounding Sphere

### 3.4 Index Structure

The advantages of applying quantization in the CSR<sup>+</sup>-tree are:

1. Assuming that  $b_c$  is fixed, the representation size of a QBS is about half that of a QBR.
2. Quantization (unlike dimensionality reduction) reduces the representation size of an entry in a predictable way, independent of the characteristics of the data sets.
3. Computing the distance between the query point and a QBS is faster than computing the distance between the query point and a QBR in high-dimensional spaces.
4. From a single computation of the distance between a query point and the centroid of a QBS, the minimum and maximum distances from the query point to the QBS are obtained by subtracting and adding the radius of the QBS. In contrast, the minimum and maximum distances from the query point to a VBR can be obtained only by two separate distance calculations.

The CSR<sup>+</sup>-tree also uses Quantized Bounding Rectangles (QBRs), which are defined as follows. In a  $d$ -dimensional space, let  $A = (\vec{a}, \vec{a}')$  be a bounding rectangle on a set of points and let  $B = (\vec{b}, \vec{b}')$  be a bounding rectangle contained in  $A$ . Note that the set of points bounded by  $B$  is a proper subset of those bounded by  $A$ . In our definition,  $A$  and  $B$  could be any bounding rectangles. However, in the CSR<sup>+</sup>-tree (or the A-tree), bounding rectangle  $B$  is a minimum bounding rectangle, and bounding rectangle  $A$  is  $B$ 's

parent minimum bounding rectangle.  $\text{QBR}(B, A) = (\vec{r}, \vec{r}')$ , the QBR of  $B$  with respect to  $A$ , is defined as follows:

$$r_i = a_i + \frac{(a'_i - a_i) \cdot u_i}{2^{b_c}}$$

where

$$u_i = \begin{cases} 2^{b_c} - 1 & b_i = a'_i \\ \lfloor (\frac{b_i - a_i}{a'_i - a_i}) \cdot 2^{b_c} \rfloor & (o.w.) \end{cases}$$

Similarly,  $r'_i = a_i + \frac{(a'_i - a_i) \cdot (u'_i + 1)}{2^{b_c}}$ , where

$$u'_i = \begin{cases} 0 & (b'_i = a_i) \\ \lceil (\frac{b'_i - a_i}{a'_i - a_i}) \cdot 2^{b_c} \rceil - 1 & (o.w.) \end{cases}$$

Intuitively,  $B$  is a quantized bounding rectangle that is expressed relative to  $A$  in a way that allows for a concise representation. Then  $\forall i, 0 \leq i < d, a_i \leq u_i \leq b_i \leq b'_i \leq (u'_i + 1) \leq a'_i$ . Note also that, since  $u_i \in [0, 2^{b_c} - 1]$ , and  $u'_i \in [0, 2^{b_c} - 1]$ , the  $\text{QBR}(B, A)$  can then be represented by a bit vector of length of  $2d \cdot b_c$  bits.

## 4 CSR<sup>+</sup>-tree

Based on the discussion and the introduction of QBSs in Section 3, we now present a new index structure, called CSR<sup>+</sup>-tree.

As shown in Figure 2, the CSR<sup>+</sup>-tree consists of index nodes and data nodes. Index nodes are further classified into QBS nodes, QBR nodes, and leaf nodes. Insertion nodes are also present in the CSR<sup>+</sup>-tree, one associated with each internal node. The configuration of each type of node is described below:

**Data Node:** A data node contains a list of data points  $(P_1, P_2, \dots, P_m)$ , where  $m$  is the number of entries.

**Leaf Node:** There is an one-to-one correspondence between data nodes and leaf nodes (4, 5, 6, and 7 in Figure 2). A leaf node corresponding to a data node  $N(P_1, P_2, \dots, P_m)$  consists of:

1. a bounding rectangle  $R$ , which is the MBR bounding  $P_1, P_2, \dots, P_m$ ;
2. a list of entries,  $(ptr_i, \text{QBS}(P_i, R))$ , where  $ptr_i$  is the pointer to the data point  $P_i$ , and  $\text{QBS}(P_i, R)$  is the representation of the QBS (with respect to  $R$ ) that approximates the data point  $P_i$ .

**QBR Node:** A QBR node (2.1, 3.1 in Figure 2) consists of:

1. a bounding rectangle  $R$ , which is the MBR bounding all MBRs in its child nodes;
2. a list of entries,  $(ptr_i, \text{QBR}(N_{c_i}, R))$ , where  $ptr_i$  is the pointer to the  $i$ -th child node  $N_{c_i}$ , which must be a leaf node, and  $\text{QBR}(N_{c_i}, R)$  is the representation of the QBR (with respect to  $R$ ) that approximates the MBR bounding all data points in  $N_{c_i}$ .
3. a pointer to an insertion node associated with this QBR node, in which there is a list of entries. The  $i$ -th entry corresponds to the  $i$ -th entry in the QBR node and it is a 2-tuple  $(\omega_i, P_{centroid_i})$ , where  $\omega_i$  is the number of data points contained in the sub-tree rooted at  $N_{c_i}$ , and  $P_{centroid_i}$  is the centroid of all data points contained in the sub-tree rooted at  $N_{c_i}$ .  $P_{centroid_i}$  is represented in  $32d$  bits, as it is in the A-tree.

**QBS Node:** A QBS node (1.1 in Figure 2) consists of:

1. a bounding rectangle  $R$ , which is the bounding rectangle of the centroids contained in the insertion node associated with this QBS node.
2. a list of entries,  $(ptr_i, \text{QBS}(N_{c_i}, R))$ , where  $ptr_i$  is the pointer to the  $i$ -th child node  $N_{c_i}$ , which is either a QBS node or a QBR node, and  $\text{QBS}(N_{c_i}, R)$  is the representation of the QBS (with respect to  $R$ ) approximating the BS bounding all data points in  $N_{c_i}$ .
3. a pointer to an insertion node associated with this QBS node, in which there is a list of entries. The  $i$ -th entry corresponds to the  $i$ -th entry in the QBS node and it is a 2-tuple  $(\omega_i, P_{centroid_i})$ , where  $\omega_i$  is the number of data points contained in the sub-tree rooted at  $N_{c_i}$ , and  $P_{centroid_i}$  is the centroid of all centroids contained in the insertion node associated with  $N_{c_i}$ . Note that the bounding sphere that is approximated by  $\text{QBS}(N_{c_i}, R)$  is centered at  $P_{centroid_i}$ . Thus,  $\text{QBS}(N_{c_i}, R)$  is obtained by substituting coordinates of  $P_{centroid_i}$  and coordinates of  $R$  into Equations (1) and (2).

**Insertion Node:** Each QBS node (or QBR node) is associated with an *insertion node* (1.2, 2.2 and 3.2 in Figure 2), which consists of  $\omega$ 's and  $P_{centroid}$ 's. The information in the insertion nodes is used only for the centroid-based insertion algorithm and updating QBSs.

#### 4.1 Computing Distance without Decompression

For disk-based index structures, quantization is used to reduce the I/O cost, which generally constitutes a major part of the query processing cost. For main memory indexing, on the other hand, besides decreasing TLB miss penalties (similar to I/O costs in disk-based indexes), our main task is to reduce the L2 cache misses and minimize the distance computation time. In existing index structures utilizing quantization, the representation of a QBS has to be decompressed before the distance between the query point and the QBS can

**Algorithm KNNSearch(point  $q$ , integer  $k$ )**

```

1. PriorityQueue  $Q$ ;
2. float  $D_1 \leftarrow \infty, D_2 \leftarrow \infty$ ;
3. List  $KNNList, KQBSList$ ;
4. enqueue( $Q, root$ );
5. while  $Q$  not empty and  $Q.MINDist \leq \min \{D_1, D_2\}$ 
6.   Node  $N \leftarrow$  dequeue( $Q$ );
7.   if  $N$  is a data point
8.     if  $dist(N, q) \leq \min \{D_1, D_2\}$ 
9.       insert( $KNNList, N, dist(N, q)$ );
10.     $D_1 \leftarrow$  getPruneDist( $KNNList$ );
11.   else
12.     point  $q' \leftarrow$  transform( $q, N.MBR$ );
13.     for each entry  $e \in N$ 
14.       if  $mindist(q', e) \leq \min \{D_1, D_2\}$ 
15.         insert( $Q, e.ptr, mindist(q', e)$ );
16.         if  $N$  is a leaf and  $maxdist(q', e.QBS) < D_2$ 
17.           insert( $KQBSList, maxdist(q', e)$ );
18.          $D_2 \leftarrow$  getPruneDist( $KQBSList$ );
19. output( $KNNList$ )

```

**Figure 3. KNN search algorithm**

be calculated. In a high-dimensional space, the decompression of a QBS representation will make the distance computation time-consuming. The search time saved by quantization might be offset by the time required to decompress the QBS. In addition, a decompressed QBS is much larger (typically by a factor of four to sixteen) than its size when compressed. This might lead to additional cache misses. To solve this problem, we propose a new distance computation algorithm that avoids the need to decompress QBS representations.

At first, a query point  $Q = \vec{q}$  is transformed into a new query point  $Q' = \vec{q}'$  with respect to the bounding rectangle  $R = (\vec{a}, \vec{a}')$  as follows:

$$q'_i = \begin{cases} \frac{(q_i - a_i)}{w_i} & (w_i > 0) \\ q_i - a_i & (w_i = 0) \end{cases} \quad (3)$$

where  $w_i = \frac{(a'_i - a_i)}{2^{bc}}$ . Then we have the following results.

**Theorem 1** *The minimum distance between  $Q$  and a QBS with respect to  $R$  can be calculated as:*

$$dist(Q, QBS) = dist(Q', \vec{v}) - r' \quad (4)$$

where  $(\vec{v}, r')$  is the representation of the QBS, and

$$dist^2(Q', \vec{v}) = \sum_{i=0}^{d-1} \left( \begin{cases} q'_i & w_i = 0 \\ w_i \cdot (q'_i - v_i - 0.5) & o.w. \end{cases} \right)^2$$

**Theorem 2** *The minimum distance between  $Q$  and a QBR with respect to  $R$  can be calculated as:*

$$dist(Q, QBR) = dist(Q', (\vec{u}, \vec{u}')) \quad (5)$$

where  $(\vec{u}, \vec{u}')$  is the representation of the QBR, and

$$dist^2(Q', (\vec{u}, \vec{u}')) = \sum_{i=0}^{d-1} \left( \begin{cases} q'_i & w_i = 0 \\ w_i \cdot (u_i - q'_i) & q'_i < u_i \\ w_i \cdot (q'_i - u'_i - 1) & q'_i > u'_i + 1 \\ 0 & o.w. \end{cases} \right)^2$$

The proofs of the theorems are omitted due to space limitations, and are available elsewhere [10].

Thus, for each index node  $N$ , using Equation (3), the query point is first transformed into a new query point with respect to the reference bounding rectangle  $N.R$ . If  $N$  is a leaf node,  $N.R$  is the MBR that bounds all data points contained in the data node corresponding to  $N$ . If  $N$  is a QBR node,  $N.R$  is the MBR bounding all MBRs in  $N$ 's child nodes. If  $N$  is a QBS node,  $N.R$  is the bounding rectangle of the centroids in the insertion node associated with  $N$ . Then, we can compute the distance between the query point and each entry contained in  $N$  using either Equation (4) or Equation (5). The transformations of the query point will take extra time during search. However, since an index node contains a large number of entries, the significant savings from avoiding the decompressions of QBSs or QBRs can more than offset of the cost of transformation of the query point.

## 4.2 Searching

Figure 3 shows the KNN search algorithm for the CSR<sup>+</sup>-tree. The algorithm is a slightly modified version of the HS algorithm [14]. We keep a priority queue  $Q$ , a  $k$ -nearest neighbors list  $KNNList$ , and a  $k$ -nearest QBSs list called  $KQBSList$ . The priority queue contains entries of nodes or data points. All entries in the queue are sorted in ascending order of MINDIST, the minimum distance between the query point and a QBS or a QBR. The  $KNNList$  keeps the  $k$ -nearest points found so far during the execution of the algorithm. It is also sorted by the distance between the query point and a data point. The  $KQBSList$  keeps the  $k$ -nearest QBSs found so far, where the QBSs approximate the data points at the leaf level. The  $KQBSList$  is sorted by the maximum distance between the query point and a QBS that approximates a data point. The queue is pruned by deleting entries whose MINDISTs to the query point exceed that to the  $k$ -th nearest data point in the  $KNNList$  or that to the  $k$ -th nearest QBS in the  $KQBSList$ .

## 4.3 Insertion

The insertion algorithm of the CSR<sup>+</sup>-tree is based on that of the SR-tree. We also adopted the centroid-based insertion algorithm because, as pointed out by Garcia-Arellano et al. [12] and Katayama et al. [16], the centroid-based insertion algorithm leads to a good clustering of data and it is suitable for nearest neighbor queries. Similar to the SS-tree and the R\*-tree, we also use the *forced reinsert* strategy to achieve dynamic reorganizations.

The strategy to determine the most suitable node to accommodate the new entry used by the CSR<sup>+</sup>-tree is to descend the tree from the root and, at each level, choose the sub-tree whose centroid is the nearest to the new entry. When an index node or a data node is full, the CSR<sup>+</sup>-tree reinserts a portion of its entries. If all those entries are reinserted into the same index node or the same data node, the node must be split. The split algorithm simply finds the dimension with the highest variance of all the centroids of its

children, and then chooses the split location to minimize the sum of the variances on each side of the split. Then, two new nodes are created, and the node closest to the original node replaces the original node. The other node is inserted.

The insertion algorithm of the CSR<sup>+</sup>-tree differs from that of the SR-tree in the way of calculating and updating QBRs and QBSs. Specifically, the CSR<sup>+</sup>-tree structure is updated as follows:

**Step 1** If  $N$  is a data node, then  $P(N)$  is the parent node of  $N$  and  $P(N).R$  is the MBR that bounds all data points contained in  $N$ . If a data point  $p$  is inserted into  $N$ , adjust  $P(N).R$ . If  $P(N).R$  is unchanged, calculate and update the QBS of  $p$ . Otherwise, update all QBSs contained in  $P(N)$ . Go to step 2 to update the parent node of  $P(N)$ .

**Step 2** If  $N$  is a QBR node, then  $P(N)$  is the parent node of  $N$  and  $N.R$  is the MBR that bounds all MBRs of  $N$ 's descendant nodes. Adjust  $N.R$  and update the centroid of the child node being updated in step 1. If  $N.R$  is unchanged, calculate and update the QBR that approximates the MBR of the child node being updated in step 1. Otherwise, update all QBRs contained in  $N$ . Go to step 3 to update the parent node of  $N$ .

**Step 3** If  $N$  is a QBS node, then  $P(N)$  is the parent node of  $N$  and  $N.R$  is the BR that bounds all centroids of  $N$ 's child nodes. Update the centroid  $c$  of the child node being updated in step 2 and adjust  $N.R$ . If  $N.R$  is unchanged, calculate and update the QBS associated with  $c$ . Otherwise, update all QBSs contained in  $N$ . If  $N$  is the root of the index tree, then stop; otherwise, go to step 3 to update  $P(N)$ .

## 5 Experimental Evaluation

We conducted an extensive experimental study to evaluate the performance of the CSR<sup>+</sup>-tree, and to compare it with existing index structures, including SS-trees, SR-trees, A-trees, and  $\Delta^+$ -trees. SS-trees, SR-trees, and A-trees were originally proposed for disk-based query processing; therefore we suitably adapted and optimized them for efficient in-memory indexing and query processing. Since the  $\Delta^+$ -tree has been shown [6] to outperform the  $\Delta$ -tree and other previous index structures (e.g., the TV-tree [18], the Slim-tree [7], the iDistance [23], the Pyramid-tree [3], and the Omni-technique [11]), we compared our approach with the  $\Delta^+$ -tree only.

Our experiments were performed on a SUN Sun Fire V440 Server with a 1.3 GHz UltraSparc 3i processor, 16GB RAM, and 1MB L2 cache with cache line size 64 bytes. The server runs Solaris 8. Experiments were also done on a Pentium 4 machine running Linux kernel 2.6.0., and similar trends were observed. Therefore, here we only show the results on the SUN platform.

In our experiments, we used two high-dimensional data sets, COREL64, and CENSUS139, with dimensionalities 64 and 139 respectively. COREL64 contains 68,040 records,

which are well clustered. CENSUS139 contains 22,784 records, and has a distribution closer to uniform [12].

For each data set, we randomly chose 1,000 records to be the query points. The results shown are averages over 1,000 queries.

Due to space limitations, only representative results are shown in this paper. A more complete set of experimental results are available elsewhere [10].

### 5.1 Selecting Node Size for the CSR<sup>+</sup>-tree

The best choice of node size for an index structure in a 1-dimensional space has been shown to be the block size of L2 cache line [19], however, this choice of the node size is not good in high-dimensional spaces. The typical block size of the L2 cache line is 64 bytes in a modern machine. Clearly, a single cache block is not sufficient to store a high-dimensional index entry. For each data set, in order to find a good, near optimal node size for the CSR<sup>+</sup>-tree, we conducted experiments in which node size varied from 1KB to 8KB. All experiments are based on 10NN queries.

Figure 4 shows the 10NN search performance of the CSR<sup>+</sup>-tree on the COREL64 data set. When the node size is small, the fan-out of the tree is also small. Thus, the number of index nodes accessed is large. From Figure 4(d), we observe that, when the node size is small, the number of TLB misses is very large. This is because the fan-out of tree is small, and small fan-out causes a large number of index nodes to be accessed during search. The rate of decrease in the number of TLB misses is initially large, but it becomes smaller as the node size increases. In the case of the larger node sizes, the fan-out of the tree is large, and more entries are packed into a node. However, the number of TLB misses decreases very slowly. Therefore, both the time for distance computation and the number of cache misses increase steadily as the node size increases beyond 3KB. Consequently, as the node size increases, search time is minimal at node size 2KB, and it increases monotonically with larger node sizes.

The results from this experiment, and from similar ones for the CENSUS139 data set, lead to the following observations: (1) There is a best choice of node size for each index structure and each data set; (2) Both the number of cache misses and the number of distance computations have significant impacts on the query performance; and (3) Unlike the case in disk-based systems, the index tree does not necessarily achieve the best performance when the number of TLB misses is at its minimum. The number of TLB misses is not as important a factor as are distance computations and cache misses. This conclusion also holds for other platforms we experimented with.

### 5.2 Comparison with the Other Indexes

In this subsection, we compare the query performance of the CSR<sup>+</sup>-tree with that of the SS-tree, the SR-tree, the A-tree, and the  $\Delta^+$ -tree. For each index structure, on each data set, we have used the node size that gives the best query

	CSR <sup>+</sup> -	A-	SS-	SR-	$\Delta^+$ -
CENSUS139	3	4	8	8	10
COREL64	2	2	8	8	6

**Table 2. Best choice of node sizes (in KB)**

performance based on extensive experiments (See Table 2 ; the word “tree” is omitted to save space.).

Although the CR-tree employs exactly the same quantization technique as in the A-tree to compress MBRs, the R-tree like partition strategy of the CR-tree produces a much worse clustering of data than the centroid-based partition strategy of the A-tree. Thus, we expect that, in general, the A-tree outperforms the CR-tree. Therefore, we did not conduct experimental comparison to CR-trees.

#### 5.2.1 Varying the Number of Nearest Neighbors

Figure 5(a) shows the average search times for the CENSUS139 data set for different values of  $K$ . The CSR<sup>+</sup>-tree is faster in search than all other index structures for all tested values of  $K$ . The CSR<sup>+</sup>-tree has the smallest number of cache misses and distance computations among all index structures (Figures 5(b)-(c)). Because the A-tree also employs a quantization technique, not surprisingly, it has a smaller number of cache misses than the SS-tree and the SR-tree. However, the A-tree has the largest number of distance computations. As a result, the A-tree has the worst performance. The SR-tree and the SS-tree spend less time on distance computations, but they have larger numbers of cache misses than the CSR<sup>+</sup>-tree and the A-tree. Although the CSR<sup>+</sup>-tree and the A-tree have smaller node sizes than the SR-tree and the SS-tree (cf. Table 2), they have larger fan-outs than the SR-tree and the SS-tree. Thus, the CSR<sup>+</sup>-tree and the A-tree have fewer index nodes being accessed during search than the SR-tree and the SS-tree. The  $\Delta^+$ -tree has by far the largest number of cache misses, mainly due to the large number of node accesses caused by extensive overlapping of bounding spheres. For example, in our experiments, when  $K=10$ , the total number of index nodes in a  $\Delta^+$ -tree is 5281, but the average number of index nodes read during the search is as large as 4901, implying that some nodes are accessed multiple times. However, overall this is well compensated by the savings in distance computations due to the use of dimensionality reduction.

In the case of the COREL64 data set (Figure 6), the CSR<sup>+</sup>-tree outperforms all other index structures for all values of  $K$ , with an average improvement of around 100% over its closest competitor, the SR-tree. The CSR<sup>+</sup>-tree has behavior similar to that in the case of the CENSUS139 data set, showing moderate increase of computation time as  $K$  increases. Again, it has the smallest number of cache misses and the least time for distance computations among all index structures. The SR-tree has much better performance than in the case of the CENSUS139 data set. This is due to the different characteristics of the two data sets. The data space for the CENSUS139 data set is sparsely populated, and the data set has a distribution that is closer to uniform, while the COREL64 data set is well clustered. The SR-tree, which uses both rectangles and spheres, is much more effec-

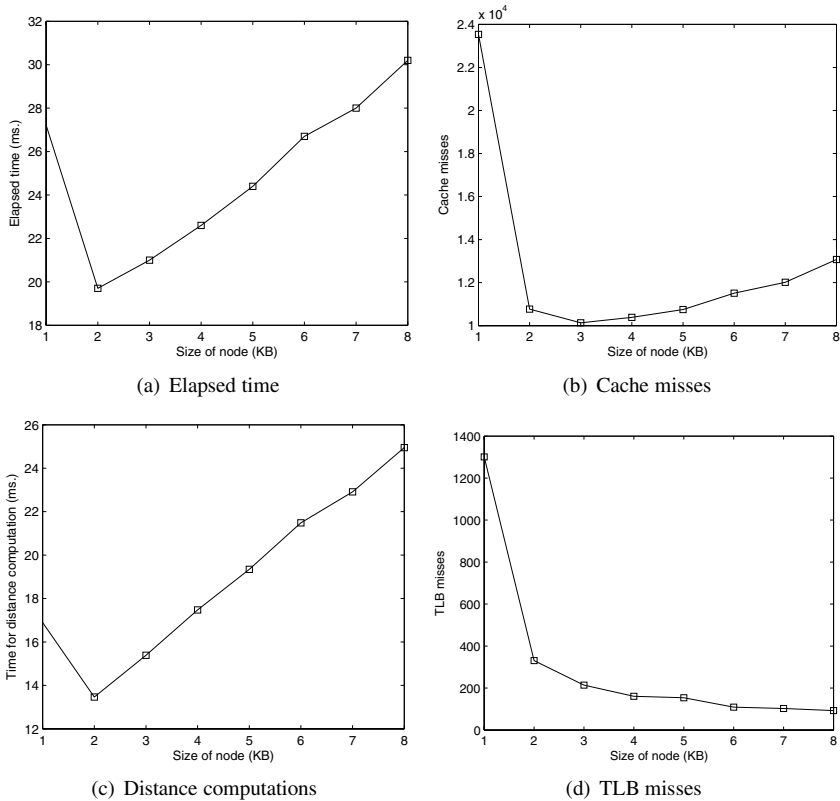


Figure 4. Effect of node size (COREL64, K=10)

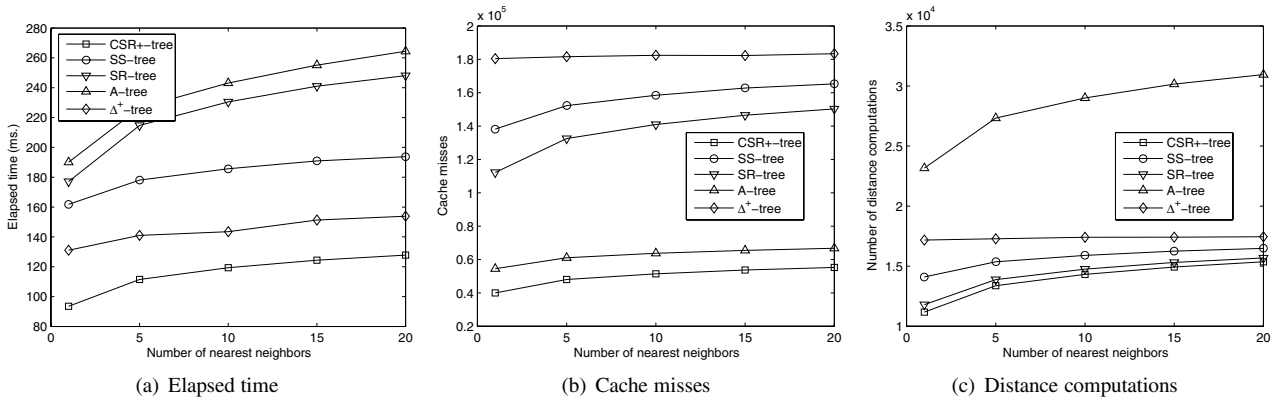


Figure 5. Effect of number of nearest neighbors (CENSUS139)

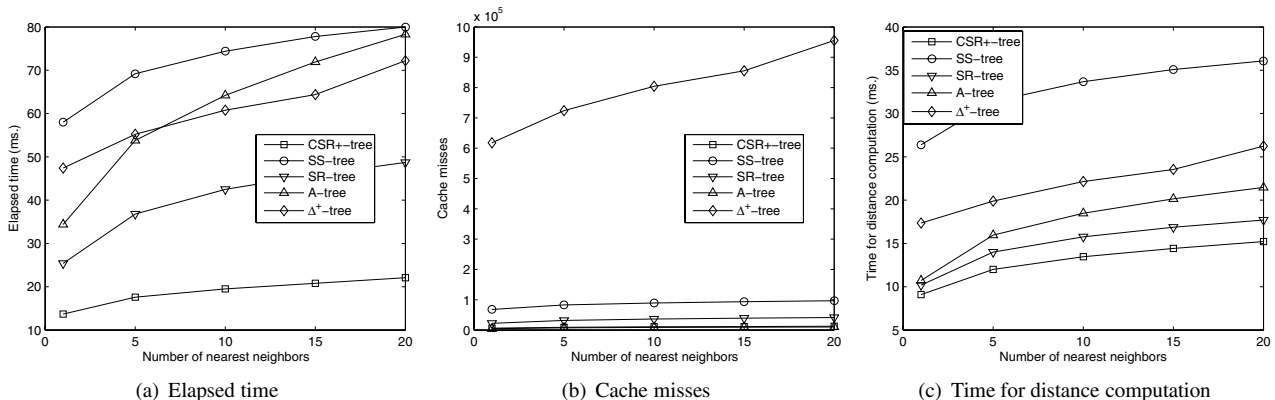


Figure 6. Effect of number of nearest neighbors (COREL64)

tive for COREL64 than for CENSUS139. The performance of both the SS-tree and the  $\Delta^+$ -tree suffers from the overlapping problem caused by the use of bounding spheres, resulting in larger number of cache misses. We observe from the experiments that for  $\Delta^+$ -tree, even if the original data form well-separated clusters, the transformed data (through PCA) may not. When the clusters in the transformed data lie close to each other, severe overlapping between the bounding spheres (of the clusters) occurs. This partly explains why the  $\Delta^+$ -tree has a much larger number of cache misses than other index structures.

It is worth noting that although other index structures have vastly different responses on the two data sets, the CSR<sup>+</sup>-tree consistently gives the best performance, which demonstrates its robustness to the varying characteristics of the data.

### 5.2.2 Varying Data Set Size

Figure 7 shows the query performance of the different index structures for subsets of different sizes of the COREL64 data set with  $K = 10$ . We took random samples of sizes ranging from 2000 to 6000 from the COREL64 data set, and evaluated the 1000 KNN queries over the samples. All index structures demonstrate similar behaviors as in the previous experiments. The CSR<sup>+</sup>-tree has the best performance in terms of elapsed time, cache misses, and time for distance computation across the range of data sizes, and has a slower rate of increase in elapsed time, validating its scalability with respect to the size of the data set. Note that to make the graph more discernible, the number of cache misses for the  $\Delta^+$ -tree is intentionally left out in Figure 7(b) as they are an order of magnitude larger than those for other structures.

### 5.2.3 Varying Dimensionality

Figure 8 shows the query performance of various index structures for different dimensionalities of the COREL64 data set with  $K = 10$ . The lower dimensionality data sets were derived from the first 8, 16, and 32 dimensions of the COREL64 data set. Again, the CSR<sup>+</sup>-tree performs consistently better than other index structures over all number of dimensions. Because the data set with a lower dimensionality is much denser than that with a higher dimensionality, not surprisingly, the SR-tree gives a performance comparable to that of CSR<sup>+</sup>-tree when the dimensionality is low. As shown in Figure 8(b) and 8(c), at dimensionalities of 8 and 16, the SR-tree has few cache misses and spends less time on distance computations. However, as the dimensionality grows, the number of cache misses of the SR-tree relative to the CSR<sup>+</sup>-tree and the A-tree increases quickly. Notice that the CSR<sup>+</sup>-tree performs well even when the dimensionality is low, although it achieves larger performance improvement over other structures when the dimensionality is higher. It can be observed from Figure 8 that CSR<sup>+</sup>-tree scales very well with respect to the number of dimensions. The A-tree has a slightly less number of cache misses than the CSR<sup>+</sup>-tree, but is significantly slower due to the amount of time spent on distance computations.

	Min	Max	Mean	Std Deviation
CSR <sup>+</sup> -tree	0.29	56.15	19.52	11.88
SS-tree	0.73	139.20	74.42	37.45
SR-tree	0.42	169.50	42.51	35.71
A-tree	0.54	283.41	64.26	59.53
$\Delta^+$ -tree	0.48	245.37	60.77	45.14

**Table 3. Distribution of elapsed time (ms)**

### 5.2.4 Query Performance Distributions

Table 3 shows the minimum, maximum, mean and standard deviation of the elapsed time for various indexes for the COREL64 data set with 1000 queries and  $K = 10$ . Observe that the mean for CSR<sup>+</sup>-tree is close to the minimum and the standard deviation is the smallest, which indicates that the distribution for the CSR<sup>+</sup>-tree is concentrated near the means. Also, the ranges between the min and max values for the CSR<sup>+</sup>-tree are small compared to other indexes. This demonstrates that the CSR<sup>+</sup>-tree has more robust and predictable performance than other index structures.

## 6 Conclusions

We presented the CSR<sup>+</sup>-tree for high-dimensional similarity search in main memory. We introduced QBSs that approximate bounding spheres and data points, in order to increase the fan-outs of index nodes and reduce the number of cache misses. The novelty of the index structure lies partly in the judicious use of both QBSs and QBRs at different levels, which takes into consideration their respective strengths and limitations. We also proposed new distance computation algorithms that avoid the need to decompress QBSs and QBRs, which help to significantly reduce the distance computation time. This is particularly useful for main memory query processing, where distance computation takes bulk of the total processing time.

We conducted extensive experiments to evaluate the CSR<sup>+</sup>-tree against several well known indexing techniques and analyzed the query performance of these techniques. Our experimental evaluation has shown that through reducing both the cache misses and distance computation time, the CSR<sup>+</sup>-tree achieves significant improvement over existing index structures, and is the best choice overall.

## Acknowledgment

This research was supported in part by an NSERC Discovery Grant and an Atkinson Minor Research Grant. The authors would like to thank our friend and mentor Prof. Kenneth C. Sevcik for his comments and encouragement during the course of this work. We miss him dearly.

## References

- [1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R\*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, 1990.

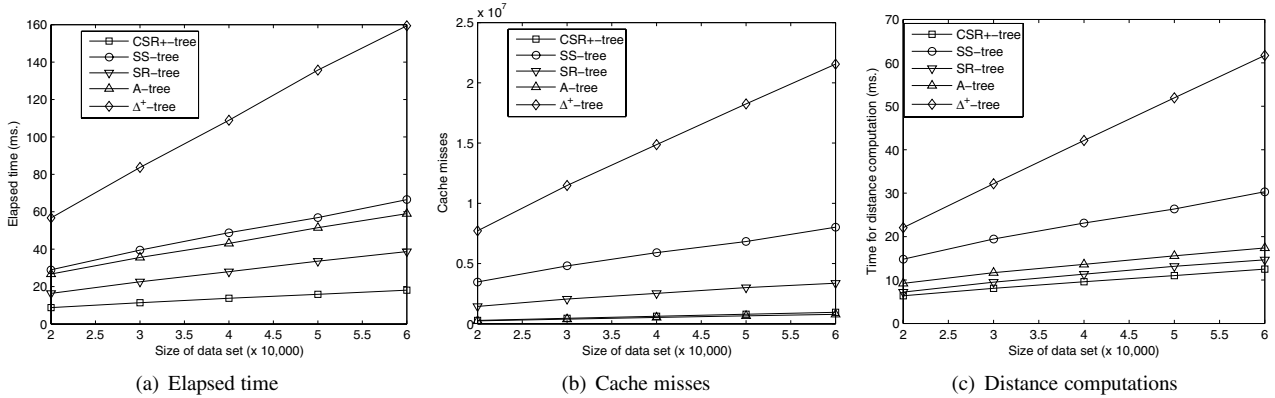


Figure 7. Effect of data set size (COREL64, K=10)

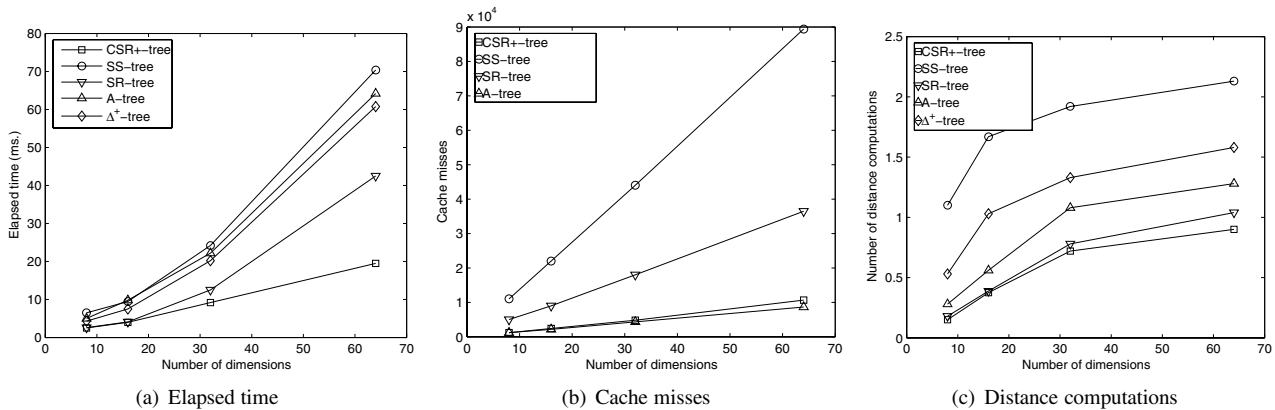


Figure 8. Effect of dimensionality (COREL64, K=10)

- [2] S. Berchtold, C. Böhm, H. V. Jagadish, H.-P. Kriegel, and J. Sander. Independent quantization: An index compression technique for high-dimensional data spaces. In *ICDE*, pages 577–588, 2000.
- [3] S. Berchtold, C. Böhm, and H.-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *SIGMOD*, pages 142–153, 1998.
- [4] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *VLDB*, pages 28–39, 1996.
- [5] T. Bozkaya and M. Özsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *SIGMOD*, pages 357–368, 2000.
- [6] B. Bui, B. C. Ooi, J. Su, and K. L. Tan. Indexing high-dimensional data for efficient in-memory similarity search. *TKDE*, 17(3), March 2005.
- [7] J. C. Traina, A. Traina, C. Faloutsos, and B. Seeger. Fast indexing and visualization of metric data sets using Slim-Trees. *TKDE*, 14(2):244–260, 2002.
- [8] B. Cui, B. Ooi, J. Su, and K. Tan. Main memory indexing: the case for BD-tree. *TKDE*, 2003.
- [9] B. Cui, B. C. Ooi, J. Su, and K. L. Tan. Contorting high dimensional data for efficient main memory KNN processing. In *SIGMOD*, pages 479–490, 2003.
- [10] J. Dong. *Indexing High-Dimensional Data for Main Memory*. MSc Thesis, Department of Computer Science, University of Toronto.
- [11] R. F. S. Filho, A. J. M. Traina, C. T. Jr., and C. Faloutsos. Similarity search without tears: The OMNI family of all-purpose access methods. In *ICDE*, pages 623–630, 2001.
- [12] C. Garcia-Arellano and K. Sevcik. Quantization techniques for similarity search in high-dimensional data spaces. In *BN-COD*, pages 75–94, 2003.
- [13] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.
- [14] G. Hjaltason and H. Samet. Ranking in spatial databases. In *SSD*, pages 83–95, 1995.
- [15] I. T. Jolliffe. *Principle Component Analysis*. Springer-Verlag, 1986.
- [16] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbour queries. In *SIGMOD*, pages 369–380, 1997.
- [17] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *SIGMOD*, pages 139–150, 2001.
- [18] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB J.*, 3(4):517–542, 1994.
- [19] J. Rao and K. A. Ross. Making B<sup>+</sup>-tree cache conscious in main memory. In *SIGMOD*, pages 475–486, 2000.
- [20] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The A-tree: An index structure for high-dimensional spaces using relative approximation. In *VLDB*, pages 516–526, 2000.
- [21] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205, 1998.
- [22] D. White and R. Jain. Similarity indexing with the SS-tree. In *ICDE*, pages 516–523, 1996.
- [23] C. Yu, B. C. Ooi, K.-L. Tan, and H. V. Jagadish. Indexing the distance: an efficient method to KNN processing. In *VLDB J.*, pages 421–430, 2001.