Information Systems 35 (2010) 825–843

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

# Indexing high-dimensional data for main-memory similarity search ☆

Xiaohui Yu [a,b,*], Junfeng Dong [c]

[a] School of Computer Science and Technology, Shandong University, Jinan, Shandong 250101, China
[b] School of Information Technology, York University, Toronto, ON, Canada M3J 1P3
[c] Microsoft Corporation, One Microsoft Way, Redmond, WA 98052–6399, United States

ARTICLE INFO

*Article history:*
Received 17 July 2009
Received in revised form
5 April 2010
Accepted 3 May 2010
Recommended by: F. Korn

*Keywords:*
High-dimensional data
Indexing
Similarity search
Cache-conscious

ABSTRACT

As RAM gets cheaper and larger, in-memory processing of data becomes increasingly affordable. In this paper, we propose a novel index structure, the CSR$^+$-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$^+$-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$^+$-tree, and compare its performance against that of other representative indexes in the literature. Our results show that the CSR$^+$-tree consistently outperforms other index structures.

© 2010 Elsevier B.V. All rights reserved.

## 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. For example, in content-based image retrieval, a core operation is similarity search, where feature vectors consist of color, texture, structure, etc. extracted from images. Such vectors are often of dimensionality 10 or even greater in many image recognition methods [19,20]. There are two common types of similarity queries: (i) $k$-nearest neighbors ($k$ NN) queries, which search for the $k$ objects from the data set that are closest to a given query point and (ii) range queries, which search for the objects that are within a specified distance to a given query point. A number of index structures have been proposed for such queries, such as the SS-tree [26] and the SR-tree [16]. Most of these index structures have largely been studied under the assumption that they are too large to fit into the main memory, and therefore the main concern is to reduce the disk I/O. However, with the rapid increase in the size of available main memory in computer systems, it is now feasible to hold many of the index structures in main memory. Consequently, the traditional bottleneck of disk accesses no longer applies in this new context.

Shifting to the main memory data processing paradigm, however, raises new issues and challenges. In particular, previous research has shown that the cache behavior has an important impact on the performance of main memory indexes. The conventional assumption that main memory accesses incur uniform cost is no longer valid in the presence of the current speed gap between cache access and main memory access. As such, it is imperative to improve the cache utilization of main memory indexes in order to achieve better performance.

☆ This paper is a substantial extension of the conference version [9].

* Corresponding author at: School of Information Technology, York University, Toronto, ON, Canada M3J 1P3.
*E-mail addresses:* xhyu@yorku.ca (X. Yu), junfeng.dong@microsoft.com (J. Dong).

0306-4379/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2010.05.001

First, it is of critical importance to minimize the L2 cache misses. Cache misses incur a substantial penalty, as the corresponding cache block must be fetched from the (much slower) main memory. Second, we have to reduce the negative impact of TLB misses on the query performance (TLB is a kind of cache that holds the translation of logical virtual memory addresses to physical page addresses for the most recently used pages.). A TLB miss occurs whenever a logical address is not found in the TLB, which incurs a significant penalty (around 100 cycles on an SUN UltraSparc CPU). Moreover, compared with disked-based indexes, distance computation now accounts for a more significant portion of the total cost of similar search, thanks to the speed difference between main memory access and disk I/O [4,7,5]. Therefore, it is also important for main memory indexes to minimize the distance computation required in performing similarity search.

A number of index structures have been proposed for main memory query processing [21,17,8], 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 [7] and $\Delta^+$−tree [5] 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$^+$-tree (CSR$^+$-tree), to support efficient similarity search in main memory. The design of the index structure centers around the notion of *quantization*. 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$^+$-tree is increased, which helps reduce the number of index nodes accessed during search. In the index structure, the QBRs are utilized in combination with 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.

Another salient feature of the CSR$^+$-tree is the drastic reduction in distance computation time by eliminating the need for decompression. In all existing quantization-based index structures, decompression (i.e., transforming the quantized representation back to the original representation) is required before distance computation. We propose a new distance computation algorithm without decompression to speed up the similarity search. The basic idea is that instead of decompressing all QBSs into the original representation in the "un-quantized" space, we transform the query into the same "quantized" space as the QBSs. Note that dimensionality reduction can also help reduce the time for distance computation due to the lower dimensionality involved, and can be employed in addition to our approach as a pre-processing step in the proposed index structure. However, we do not pursue this further in this paper as it is orthogonal to our proposal.

Our contributions in this paper 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$^+$-tree. The judicious use of both QBSs and QBRs significantly reduces the fan-out of the index structure and results in less L2 cache misses and TLB misses.
- 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 data sets of different characteristics to evaluate the proposed index structure. We also performed a thorough comparative study of our approach with existing index structures.

Although the proposed index structure can handle both range queries and $k$ NN queries, we will be focusing on $k$ NN queries in the sequel because of the more complex nature of $k$ NN query processing. Experimental results on range queries are also included for completeness.

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

## 2. Related work

High-dimensionality similarity search has been extensively studied in the literature, and a large number of index structures have been proposed, e.g., the X-tree [2], VA-file [25], IQ-tree [1], A-tree [23], iDistance [13], and LSB-tree [24]. In these proposals, one or more of the following three techniques are employed, namely, hierarchical organization, quantization, and dimensionality reduction. Various indexing structures are proposed based on the use of combinations of these techniques. High-dimensional index structures have also been proposed for the related problem of similarity join (e.g., [15]). In what follows, we will first briefly review the SS-tree, the SR-tree, and the A-tree, which has inspired our proposal of the CSR$^+$-tree, and then describes related work in main memory index structures.

### 2.1. High-dimensional index structures

The SS-tree [26] is a hierarchical index structure that employs bounding spheres rather than bounding rectangles as page regions to organize the points. For maintenance efficiency, the bounding spheres are not minimum bounding spheres. Rather, the center of a bounding sphere is the centroid of all points bounded by the sphere, and the radius is sufficiently large to include all the points in the sphere. When inserting a point, the insertion algorithm chooses a subtree whose centroid is the nearest to the point. When an overflow occurs in a node, 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. According to the analysis presented by White and Jain, bounding spheres are better suited for processing $k$ NN queries using the $L_2$ metric than are bounding rectangles. However, a big drawback of bounding spheres is that a bounding sphere tends to have a much larger volume than the corresponding bounding rectangle in high-dimensional spaces. Due to the high degree of overlap between bounding spheres, the performance of the SS-tree deteriorates quickly as dimensionality grows.

Katayama and Satoh observed that the average volume of a bounding rectangle is much smaller than that of a bounding sphere, and that the average diameter of a bounding rectangle is much larger than that of a bounding sphere. Therefore, they proposed the SR-tree [16], which uses the intersection solid between a bounding rectangle and a bounding sphere as the page region.

The SR-tree applies the centroid based insertion algorithm of the SS-tree. When searching for the nearest neighbor, both the bounding rectangle and the bounding sphere are used, which provides a better estimation of the distance from a query point to the nearest point in a region. Due to the strong pruning ability offered by this combination, the SR-tree outperforms both the R*-tree and the SS-tree in high-dimensional spaces. However, each entry of a node of the SR-tree includes $(3d+1)$ values, where $d$ is the dimensionality of the data. As dimensionality grows, the fan-out of the SR-tree decreases very quickly compared to the R*-tree and the SS-tree, resulting in deterioration in query performance.

The A-tree, proposed by Sakurai et al. [23], is a hierarchical index structure, derived from the SR-tree and the VA-file. The basic idea of the A-tree is the introduction of virtual bounding rectangles (VBRs) that approximate MBRs (minimum bounding rectangles) or vectors. In the A-tree, a child MBR is compressed as a VBR relative to its parent MBR using a few bits per dimension, such that the child MBR is fully contained within the hyper-rectangle specified by the VBR. Because VBRs can be represented rather compactly, tree nodes can contain a large number of VBR entries so that fan-out is large, which leads to fast search.

The authors also observed that, in high-dimensional spaces, BSs have much larger volumes than MBRs, thus, the frequency of using BSs decreases as dimensionality increases. On the other hand, the centroid based partitioning strategy produces a good clustering of the data set. In order to support insertions while keeping internal node fan-out high, the data structure is augmented by *insertion nodes*. In each internal node, the VBRs are grouped together and placed in the associated insertion node. Insertion nodes are not accessed during searches. The bigger fan-out and the good clustering of data enable the A-tree to have competitive performance. The experiments conducted by Sakurai et al. show that the A-tree outperforms the SR-tree and the VA-file.

### 2.2. Cache conscious index structures

The index structures described in the preceding subsection are proposed in the context of disk-based similarity search, where the data and index structures are assumed to be too large to fit in memory. This assumption is increasingly being challenged as random access memory (RAM) gets cheaper and larger, and main memory indexing has attracted a great deal of interest.

Rao and Ross [21] were among the first in addressing the importance of cache behavior in the design of main memory indexes, and proposed two main memory index structures, CSS-tree [22] and CSB$^+$-tree [21], that are designed to be cache conscious. Both indexes adopt a pointer elimination technique that effectively doubles the fan-out of the tree. It thus allows good utilization of cache lines and significantly reduces L2 cache misses. However, the two indexes are only targeted at single dimensional data. The pointer elimination technique does not benefit high-dimensional index structures, where the size of a pointer is far smaller than the size of an entry such as an MBR or a data object.

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 [7].

Cui et al. presented the $\Delta$- tree [7], a multi-level main memory index structure in which each level represents the data space with different dimensionalities, such that the root level uses the least number of dimensions and the leaf level contains data at their full dimensionality. The $\Delta$- tree employs a dimensionality reduction technique, principal component analysis (PCA) [14], in its indexing scheme.

The $\Delta$- tree fixes the fan-out for all levels of the tree. Given a set of $N$ points and a fan-out $f$, the maximum number of levels of the $\Delta$- tree is defined as $L = \lceil \log_f N \rceil$. To determine the number of dimensions $m_l$ to be used at level $l$, a heuristic selects the smallest $m_l$ such that the percentage of variation accounted for by the first $m_l$ dimensions is greater or equal to $l/L, 1 \le l \le L$. Thus, the number of dimensions $m_l$ to be used at level $l$ is

determined by

$$m_l = \min_k \left\{ \frac{\sum_{j=1}^{k} var_j}{\sum_{i=1}^{d} var_i} \geq \frac{l}{L} | 1 \leq l \leq L \right\}$$

where $var_i$ is the variance of the positions of the points in dimension $i$.

The $\Delta$- tree first transforms all data points in the original $d$-dimensional space into the data points in the new $d$-dimensional space by applying PCA. It then uses the $k$-means algorithm to partition the data into clusters using only the first $m_l$ dimensions associated with the level $l$. Since dimensionality reduction is used, the cost of distance computations is decreased, and the fan-out of the tree is enlarged. Their study showed that the $\Delta$- tree outperforms other well-known schemes in main memory.

The $\Delta^+$−tree [5] 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. Moreover, since few dimensions in the higher levels of the tree are used to cluster the data, the loss of information can lead to a bad clustering. The bad clustering of data impairs the performance of the $\Delta$- tree.

## 3. Motivation for the CSR$^+$-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 (assuming the nodes are full): $T_{dist} = N \cdot (S/d \cdot B) \cdot t_{dist}$, 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: $T_{cache} = N \cdot (S/c) \cdot t_{cache}$, 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, $N \cdot S/d \cdot B$, (2) the total number of cache misses, $N \cdot S/c$, (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

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

where $b_{leaf}$ is the number of bytes per dimension for a leaf node entry, $b_{non\text{-}leaf}$ 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.

**Table 1**
Notations.

| | |
|---|---|
| $d$ | 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 |

### 3.2. BS, BR and BS+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 [23].)

### 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 (with the number of reduced dimensions set experimentally for specific applications), and then build an index structure over the reduced data set.

A QBS (Fig. 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

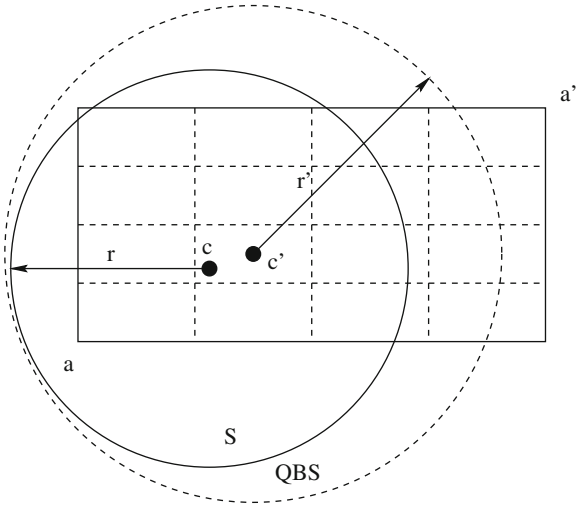



**Fig. 1.** Quantized bounding sphere.

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:

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

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

where

$$v_i = \begin{cases} \left\lfloor \left(\dfrac{c_i - a_i}{a'_i - a_i}\right)\cdot 2^{b_c} \right\rfloor, & a_i \leq c_i < a'_i \\ 2^{b_c}-1, & c_i = a'_i \end{cases} \tag{2}$$

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.

The advantages of applying quantization in the CSR$^+$-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$^+$-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$^+$-tree (or the A-tree), bounding rectangle $B$ is a minimum bounding rectangle, and bounding rectangle $A$ is $B$'s parent minimum bounding rectangle. QBR $(B,A)=(\vec{r},\vec{r'})$, the QBR of $B$ with respect to $A$, is defined as follows:

$$r_i = a_i + \frac{(a'_i - a_i)\cdot u_i}{2^{b_c}} \tag{3}$$

where

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

Similarly,

$$r'_i = a_i + \frac{(a'_i - a_i)\cdot (u'_i+1)}{2^{b_c}} \tag{4}$$

where

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

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

## 4. CSR$^+$-tree

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

### 4.1. Index structure

As shown in Fig. 2, the CSR$^+$-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$^+$-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 $(P_1, P_2, \ldots, P_m)$, 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 Fig. 2). A leaf node corresponding to a data node $N(P_1, P_2, \ldots, P_m)$ consists of:

(1) a bounding rectangle $R$, which is the MBR bounding $P_1, P_2, \ldots, P_m$;
(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 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 Fig. 2) consists of:

(1) a bounding rectangle $R$, which is the MBR bounding all MBRs in its child nodes;
(2) a list of entries, $(ptr_i, \text{QBR}(Nc_i, R))$, where $ptr_i$ is the pointer to the $i$-th child node $Nc_i$, which must be a leaf node, and QBR$(Nc_i, R)$ is the representation of the QBR (with respect to $R$) that approximates the MBR bounding all data points in $Nc_i$;
(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 $Nc_i$, and $P_{centroid_i}$ is the centroid of all data points contained

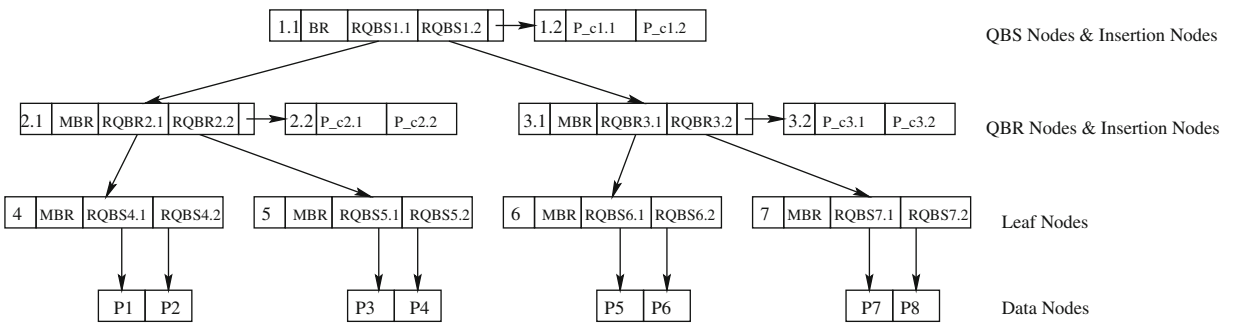



**Fig. 2.** Structure of the CSR$^+$-tree.

in the sub-tree rooted at $Nc_i$. $P_{centroid_i}$ is represented in $32d$ bits, as it is in the A-tree.

*QBS node*: A QBS node (1.1 in Fig. 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, ($ptr_i$, QBS($Nc_i$, R)), where $ptr_i$ is the pointer to the $i$-th child node $Nc_i$, which is either a QBS node or a QBR node, and QBS($Nc_i$, $R$) is the representation of the QBS (with respect to $R$) approximating the BS bounding all data points in $Nc_i$;
(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 $Nc_i$, and $P_{centroid_i}$ is the centroid of all centroids contained in the insertion node associated with $Nc_i$. Note that the bounding sphere that is approximated by QBS($Nc_i$, $R$) is centered at $P_{centroid_i}$. Thus, QBS($Nc_i$, $R$) is obtained by substituting coordinates of $P_{centroid_i}$ and coordinates of $R$ into Eqs. (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 Fig. 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.

An illustration of the CSR$^+$-tree (based on Fig. 2) is presented in the Appendix to help the readers better understand the index structure.

### 4.2. 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 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:

$$q'_i=\begin{cases}\dfrac{(q_i-a_i)}{w_i} & (w_i>0)\\ q_i-a_i & (w_i=0)\end{cases} \tag{5}$$

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

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

$$dist(Q,QBS)=dist(Q',\vec{v})-r' \tag{6}$$

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

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

**Proof.** Let the QBS be $(\vec{c},r')$, where

$$c_i=a_i+w_i\cdot(v_i+0.5) \tag{7}$$

The distance between the query point $Q$ and $\vec{c}$ can be evaluated as

$$\mathrm{dist}^2(\vec{q},\vec{c})=\sum_{i=0}^{d-1}X_i^2=\sum_{i=0}^{d-1}(q_i-c_i)^2 \tag{8}$$

Now, substitute (7) in (8). If $w_i$=0, we have

$X_i=(q_i-a_i)=q'_i$

Otherwise, we have

$X_i=q_i-a_i-w_i(v_i+0.5)$ and

$q'_i=\dfrac{(q_i-a_i)}{w_i}\Longrightarrow X_i=w_i\cdot(q'_i-v_i-0.5)$ □

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

$$\mathrm{dist}(Q,\mathrm{QBR})=\mathrm{dist}(Q',(\vec{u},\vec{u}')) \tag{9}$$

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

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

**Proof.** Let the QBR be $(\vec{r},\vec{r}')$, which is given by

$$\{(a_i+w_i\cdot u_i),(a_i+w_i\cdot(u'_i+1))|0\le i<d\} \tag{10}$$

The distance between the query point and the QBR can be evaluated as

$$\mathrm{MINDIST}^2(q,\mathrm{QBS})=\sum_{i=0}^{d-1}X_i^2=\sum_{i=0}^{d-1}\left(\begin{cases}r_i-q_i, & q_i<r_i\\ q_i-r'_i, & q_i>r'_i\\ 0, & o.w.\end{cases}\right) \tag{11}$$

Now, substitute (10) into (11). If $w_i$=0, we have

$X_i^2=(q_i-a_i)^2=q'_i2$

Otherwise, we have

$$X_i = \begin{cases} a_i + w_i \cdot u_i - q_i, & q_i < a_i + w_i \cdot u_i \\ q_i - a_i - w_i \cdot (u'_i + 1), & q_i > a_i + w_i \cdot (u'_i + 1) \\ 0, & o.w. \end{cases}$$

and since $q'_i = (q_i - a_i)/w_i$, we have

$$X_i = \begin{cases} w_i \cdot (u_i - q'_i), & q'_i < u_i \\ w_i \cdot (q'_i - u'_i - 1), & q'_i > (u'_i + 1) \\ 0 & o.w. \end{cases} \quad \square$$

Thus, for each index node $N$, using Eq. (5), 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 contained 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 Eq. (6) or (9). 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.

### 4.3. Search

Fig. 3 shows the $k$ NN search algorithm for the CSR$^+$-tree. The algorithm is a slightly modified version of the HS algorithm [12]. 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*.

```
Algorithm kNN Search (point q, integer k)
1. Priority Queue Q;
2. float D1 ← ∞ , D2 ← ∞ ;
3. List KNNList , KQBSList ;
4. enqueue( Q, root);
5. while Q not empty and Q.MINDist ≤ min {D1, D2}
6.    Node N ← dequeue (Q);
7.    if N is a data point
8.       if dist (N, q) ≤ min {D1, D2}
9.          insert( KNNList, N, dist (N, q));
10.         D1 ← getPruneDist( KNNList );
11.   else
12.      point q' ← transform (q, N.MBR);
13.      for each entry e ∈ N
14.         if mindist (q', e) ≤ min {D1, D2}
15.            insert( Q, e.ptr, mindist (q', e));
16.            if N is a leaf and maxdist (q', e.QBS) < D2
17.               insert( KQBSList, maxdist (q', e));
18.               D2 ← getPruneDist( KQBSList);
19.output ( KNNList )
```

**Fig. 3.** $k$ NN search algorithm.

### 4.4. Insertion

The insertion algorithm of the CSR$^+$-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. [11] 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$^+$-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$^+$-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$^+$-tree differs from that of the SR-tree in the way of calculating and updating QBRs and QBSs. Specifically, the CSR$^+$-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 $P(N)$.

### 4.5. Discussion

The CSR$^+$-tree has the following advantages for supporting the data set search in main memory:

(1) The CSR$^+$-tree takes less space than the A-tree or the SR-tree.
(2) Given a fixed code length, the representation size of a QBS is roughly half that of a VBR. Thus in higher levels of the index tree, the fan-out of the CSR$^+$-tree is larger than that of the A-tree.
(3) In contrast to the $\Delta^+$- tree, the CSR$^+$-tree is also suitable for data sets not amenable to PCA-based dimensionality reduction.
(4) Unlike the $\Delta^+$- tree and the IQ-tree, it is a truly dynamic index structure.

## 5. Experimental evaluation

We conducted an extensive experimental study to evaluate the performance of the CSR$^+$-tree, and to compare it with existing index structures, including SS-trees, SR-trees, A-trees, and $\Delta^+$−tree s. 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 [5] to outperform the $\Delta$- tree and other previous index structures (e.g., the TV-tree [18], the Slim-tree [6], the iDistance [27], the Pyramid-tree [3], and the Omni-technique [10]), 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, 16 GB RAM, and 1 MB 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 [11].

For each data set, we randomly chose 1000 records to be the query points. The results shown are averages over 1000 queries.

### 5.1. Selecting node size for the CSR$^+$-tree

The best choice of node size for an index structure in a one-dimensional space has been shown to be the block size of L2 cache line [21], 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$^+$-tree, we conducted experiments in which node size varied from 1 to 8 kB. All experiments are based on 10NN queries.

Fig. 4 shows the 10NN search performance of the CSR$^+$-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 Fig. 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 3 kB. Consequently, as the node size increases, search time is minimal at node size 2 kB, 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. Effect of quantization on the CSR$^+$-tree

One of the key innovations of the CSR$^+$-tree is the use of quantization, which helps reduces the representation sizes of the MBRs and BSs. In order to study the effect of quantization on the performance of the CSR$^+$-tree, we conducted experiments with varying quantization levels, with all other parameters fixed. Note that 32 bits are dedicated to each dimension in the full (unquantized) representation of the data points, MBRs, and BSs. In the experiments, we varied the number of bits used to represent each dimension ($b_c$) in the QBRs and QBSs from 4 (corresponding to a compression ratio of 8) to 32 (the unquantized representation). The results on performing 10NN search on the COREL64 data set is reported in Fig. 5.

When more bits are used for each dimension (less quantization), a smaller number of entries can be fit in an index node, reducing the fan-out of the tree. This results in an increase in the number of nodes accessed during the search, which explains the increase in the number of cache misses and TLB misses with increasing $b_c$ in Fig. 5(b) and (d). The case for the number of distance computations is more complicated. It is directly proportional to the number of index entries (not the index nodes) accessed,

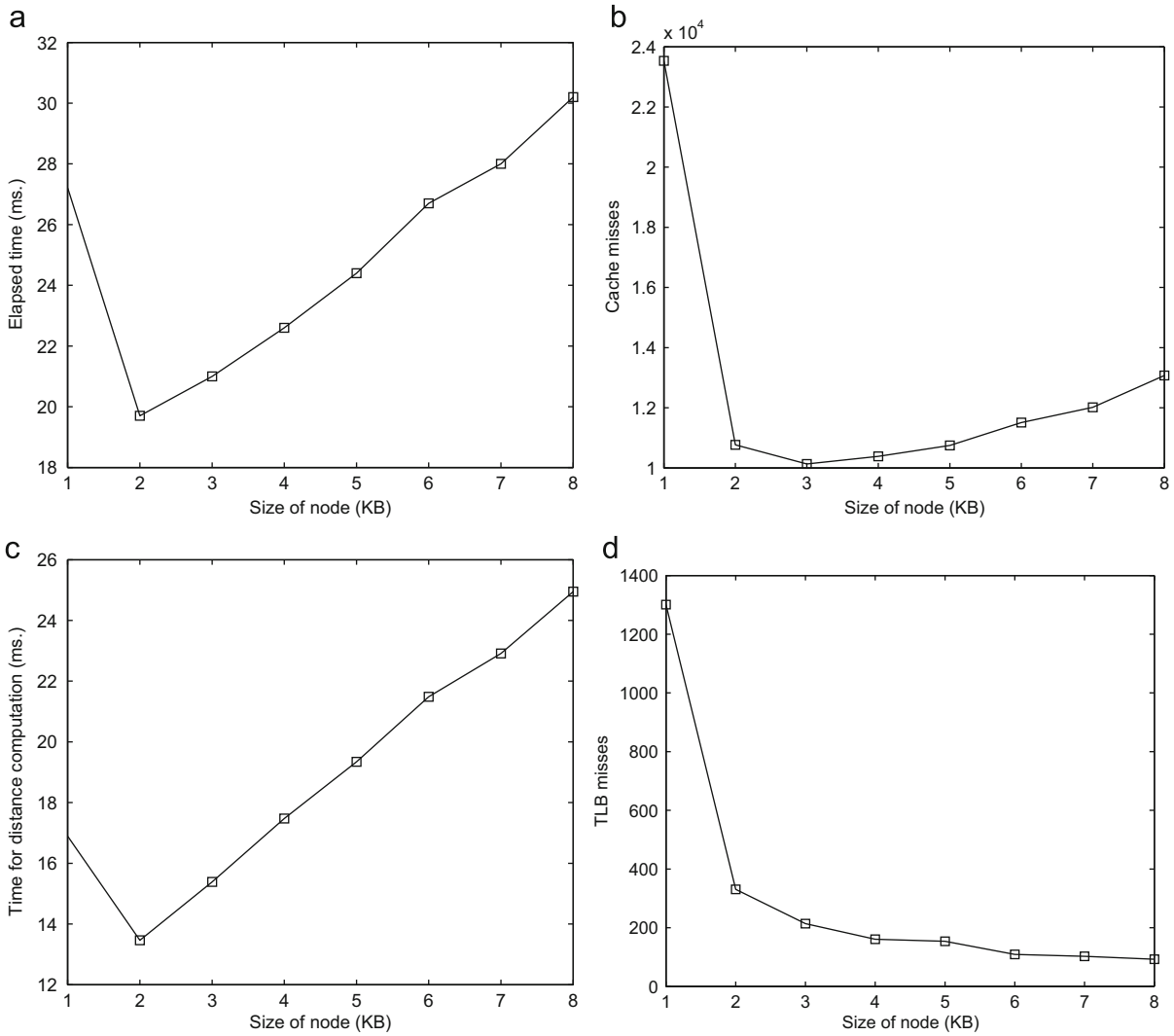



**Fig. 4.** Effect of node size (COREL64, $K$=10): (a) elapsed time; (b) cache misses; (c) distance computations; (d) TLB misses.

which is affected by two factors: the number of index nodes accessed, and the number of index entries in each node. When the level of quantization is increased (corresponding to decreasing $b_c$), the first factor decreases as we have just analyzed, while the second factor increases as more entries can be packed into each node. The interplay of the two factors results in the convex curve shown in Fig. 5(c), with the minimum obtained at $b_c$=8. Since the time for distance computation accounts for a large portion of total time for search, the curve for elapsed time in Fig. 5(a) follows a similar pattern. It is worth noting that by using 8 bits to represent each dimension, a more than twofold speedup is achieved over using the original 32-bit representation, validating the effectiveness of quantization in the CSR$^+$-tree. For each application, the right level of quantization can be determined experimentally. Our experience with the data sets we used indicate that $b_c$=8 is in general a good choice.

### 5.3. Effects of various data and query properties

In this subsection, we study the performance of the CSR$^+$-tree under different data and query properties. In particular, we evaluate how the following factors affect its performance: the number of nearest neighbors specified in the query, the number of records in the data set, and the dimensionality of the data. We also compare the query performance of the CSR$^+$-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 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

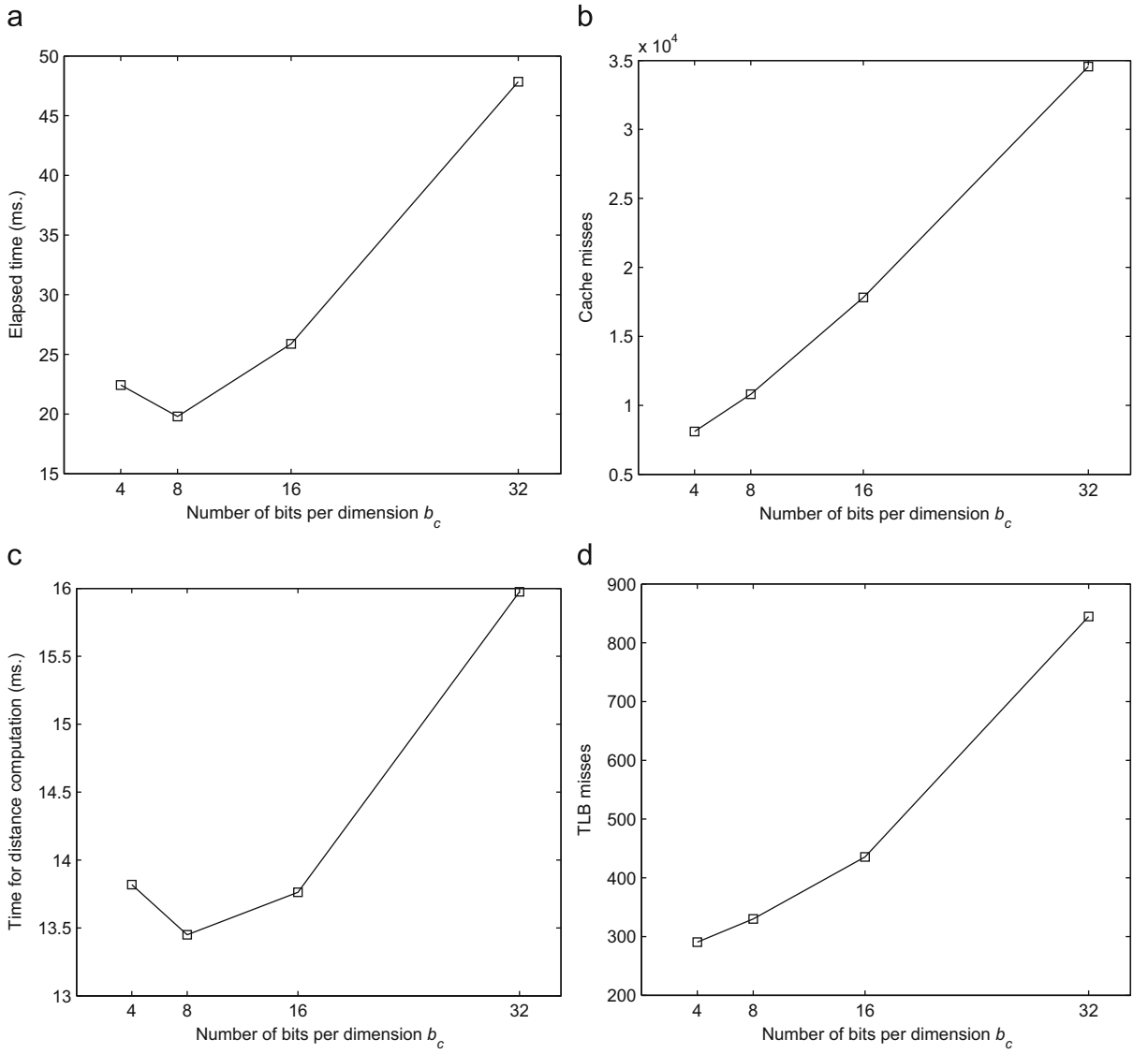



**Fig. 5.** Effect of quantization level (COREL64, $K=10$, node size=2 kB): (a) elapsed time; (b) cache misses; (c) distance computations; (d) TLB misses.

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

| | CSR$^+$-tree | A-tree | SS-tree | SR-tree | $\Delta^+$−tree |
|---|---|---|---|---|---|
| CENSUS139 | 3 | 4 | 8 | 8 | 10 |
| COREL64 | 2 | 2 | 8 | 8 | 6 |

that, in general, the A-tree outperforms the CR-tree. Therefore, we did not conduct experimental comparison to CR-trees.

*5.3.1. Varying the number of nearest neighbors*

Fig. 6(a) shows the average search times for the CENSUS139 data set for different values of $K$. The CSR$^+$-tree is faster in search than all other index structures for all tested values of $K$. The CSR$^+$-tree has the smallest number of cache misses and distance computations among all index structures (Fig. 6(b) and (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$^+$-tree and the A-tree. Although the CSR$^+$-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$^+$-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

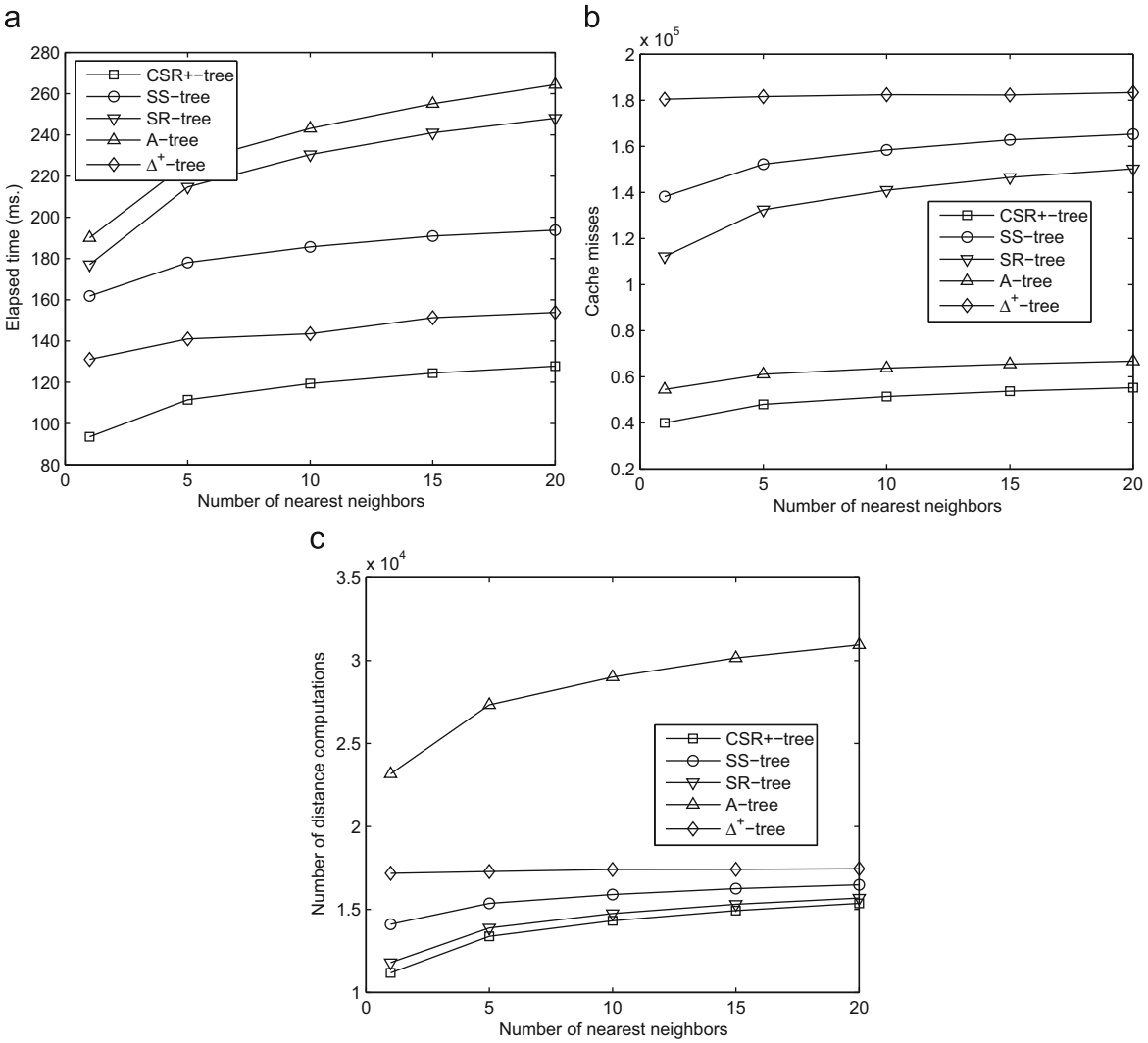



**Fig. 6.** Effect of number of nearest neighbors (CENSUS139): (a) elapsed time; (b) cache misses; (c) distance computations.

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 (Fig. 7), the CSR$^+$-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$^+$-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 effective 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

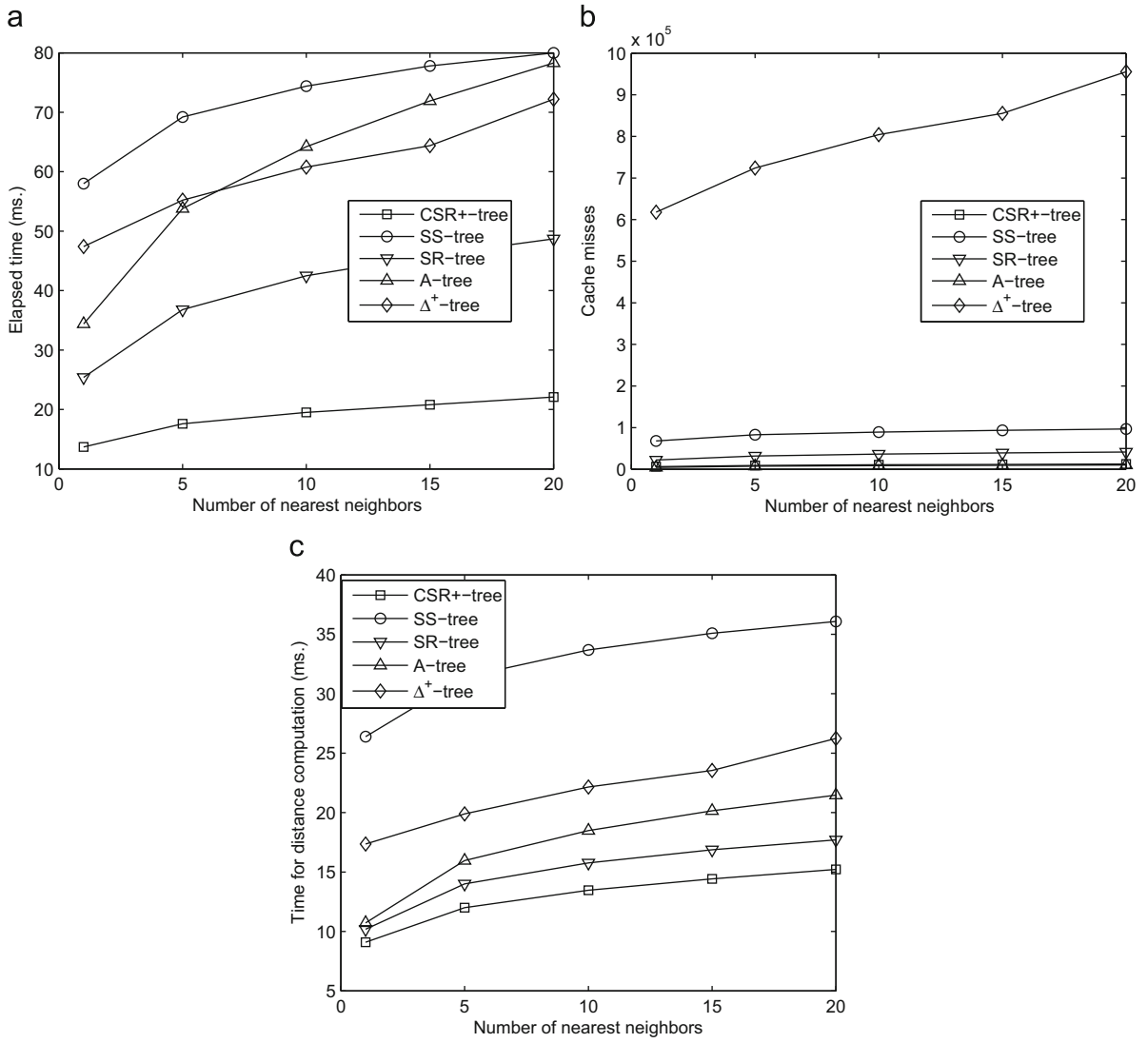



**Fig. 7.** Effect of number of nearest neighbors (COREL64): (a) elapsed time; (b) cache misses; (c) times for distance computation.

CSR$^+$-tree consistently gives the best performance, which demonstrates its robustness to the varying characteristics of the data.

### 5.3.2. *Varying data set size*

Fig. 8 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 $k$ NN queries over the samples. All index structures demonstrate similar behaviors as in the previous experiments. The CSR$^+$-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.

Since the COREL64 data set consists of only 68,040 records, larger data sets are needed in order to further test the scalability of the CSR$^+$-tree. Due to the lack of real high-dimensional data sets of sufficiently large sizes, we synthesized a collection of uniformly distributed data sets of dimensionality 60, with the number of records (data points) in each data set ranging from 1000 to 1,000,000. We evaluated 1000 randomly generated 10-NN queries over the data sets. The results in terms of elapsed time are shown in Fig. 9(a). Note that the plot is in log–log scale. For clarity, we also present the same results numerically in Fig. 9(b). It is evident that the CSR$^+$-tree scales almost linearly with respect to the data size.

### 5.3.3. *Varying dimensionality*

Fig. 10 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$^+$-tree performs

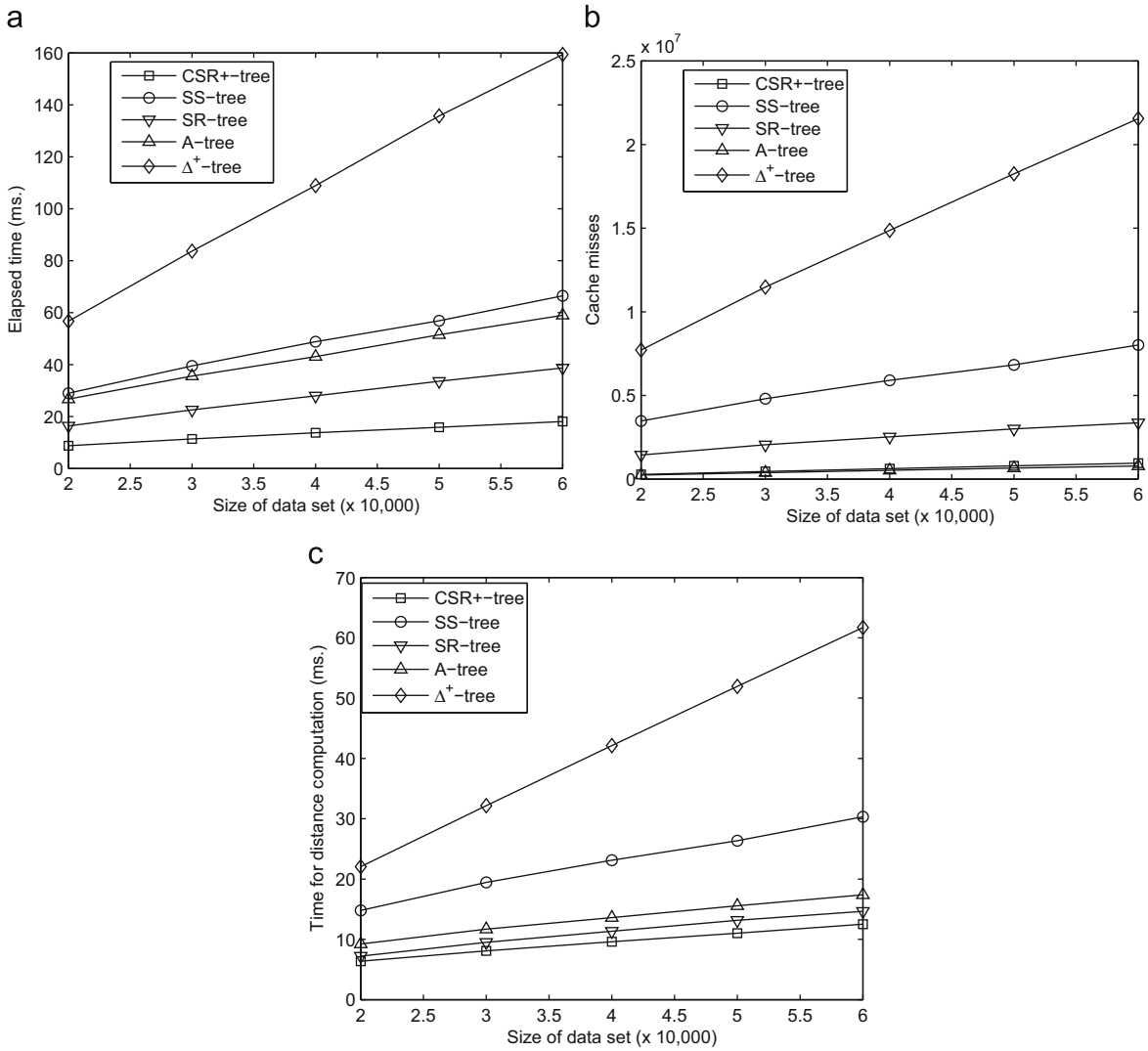



**Fig. 8.** Effect of data set size (COREL64, $K$=10): (a) elapsed time; (b) cache misses; (c) distance computations.

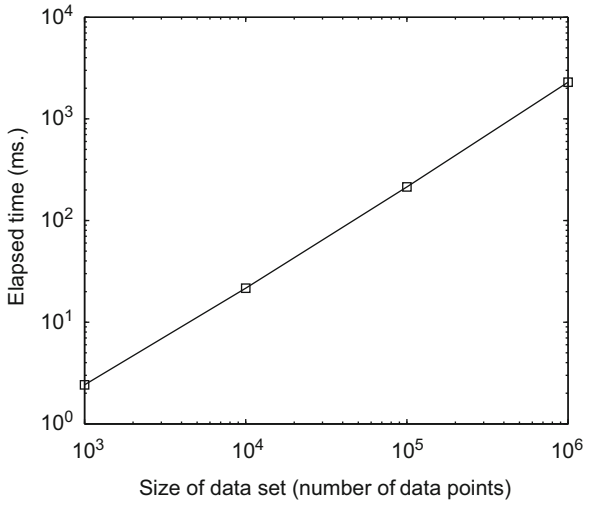
consistently better than other index structures over all number of dimensions. Because the data set with a lower dimensionality is much denser than that with a higher dimensionality, not surprisingly, the SR-tree gives a performance comparable to that of CSR$^+$-tree when the dimensionality is low. As shown in Fig. 10(b) and (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$^+$-tree and the A-tree increases quickly. Notice that the CSR$^+$-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 Fig. 10 that CSR$^+$-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$^+$-tree, but is significantly slower due to the amount of time spent on distance computations. Note that to make the graph more discernible, the number of cache misses for the $\Delta^+$−tree is intentionally left out in Fig. 10(b) as they are an order of magnitude larger than those for other structures.

### 5.3.4. *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$^+$-tree is close to the minimum and the standard deviation is the smallest, which indicates that the distribution for the CSR$^+$-tree is concentrated near the means. Also, the ranges between the min and max values for the CSR$^+$-tree are small compared to other indexes. This demonstrates that the CSR$^+$-tree has more robust and predictable performance than other index structures.

Interestingly, the A-tree yields the largest maximum and standard deviation of the elapsed time. We believe

a




b

| data size | 1,000 | 10,000 | 100,000 | 1,000,000 |
|---|---|---|---|---|
| elapsed time (s) | 0.0024 | 0.0216 | 0.2143 | 2.2951 |

**Fig. 9.** Scalability w.r.t. data set size (uniform 60-dimensional data, $K=10$): (a) a log–log plot of the elapsed time vs. data size; (b) data table.

the reason is that, the A-tree uses bounding rectangles in the internal nodes, and the average diameter of bounding rectangles is much larger than that of bounding spheres.

### 5.4. Performance on range queries

In addition to the nearest neighbor queries, we also evaluated the performance of the various index structures on another common type of similarity queries—range queries. For a given query center $P$, a range query of radius $r$ ($r > 0$) is to search for all the data points that lie within a distance (to $P$) of less than or equal to $r$. Here we report the experimental results on the COREL64 data set. Similar trends were observed on the CENSUS139 data set.

For all the tuples in the data set, we normalize their values in each dimension so that they all fall in the range [0,1]. We randomly choose 1000 tuples from the data set, and use them as query centers. We then vary the query radius $r$ from 0.05 to 0.25 for all queries, and measure the average performance of the competing index structures over the 1000 queries. The results are reported in Fig. 11. Again, the CSR$^+$-tree demonstrates superior performance over other index structures in all departments. It is interesting to observe that, because of the similar centroid-based insertion strategies used in the CSR$^+$-tree and the A-tree, the numbers of cache misses for the A-tree are close to those for the CSR$^+$-tree (Fig. 11(b)). However, as we increase the radius of the range queries, the cost for distance computations in the A-tree increases much faster than for other index structures (Fig. 11(c)) and it eventually makes the A-tree the worst performer when the radius gets past 0.15 (Fig. 11(a)). This is mainly due to the way distances are computed in A-tree—the VBRs are decompressed first, and then the distances are computed. In contrast, the distance computation time for the CSR$^+$-tree remains lower than that for other index structures, mainly due to our proposed distance computation algorithm that does not require decompression. The $\Delta^+$-tree suffers from a larger number of cache misses than other index structures when the radius is greater than 0.15. This can be attributed to the significant overlapping of bounding spheres as discussed in Section 5.3.1.

It is worth pointing out that when the radius of the range query gets large enough such that a significant portion of the tuples are included in the query result, no index structures will outperform a simple sequential scan of the tuples. In particular, in our experiments, all the index structures are beat by sequential scan when the radius is greater than 0.4.

### 5.5. Construction time

We measure the construction time of the various index structures as we vary the size of the data set, and the results are reported in Fig. 12. For each index structure, we have used the node size that gives the best performance (Table 2). The A-tree gives the best construction time, followed by the CSR$^+$-tree, and then the other index structures. The A-tree does best due to the relatively smaller leaf node fan-out and internal node fan-out (see Table 4). The smaller fan-out leads to a smaller number of reinsertions during construction. Although the CSR$^+$-tree has a similar leaf node fan-out to the A-tree, the internal node fan-out of the CSR$^+$-tree is almost two times larger than that of the A-tree. Due to a larger number of reinsertions at the higher levels, the CSR$^+$-tree has a longer construction time than the A-tree. Because it has the largest leaf node fan-out and internal node fan-out, the SS-tree has the largest number of reinsertions. Therefore, the SS-tree yields the worst construction performance when the size of the data set is less than 40,000. The $\Delta^+$-tree takes the longest to construct when the size of data becomes large, mainly due to the high cost associated with principal component analysis. Note that since the construction cost is usually amortized over many queries over the index structure, it is not as important as the query performance.

### 5.6. Disk-based performance evaluation

Although in this paper we are mainly interested in main memory indexing, the experiments we conducted also show that the CSR$^+$-tree is a good candidate for systems in which data reside on disk. In a disk-based system, the page size is the minimum number of bytes to be manipulated by an I/O operation. In the index structures that we are considering, it is also the size of the organizational units, e.g., index nodes in the CSR$^+$-tree. For the A-tree and the SR-tree, we used the same page size (8 kB) suggested by the authors of the two techniques. For the CSR$^+$-tree and the SS-tree, we also used a page size of 8 kB, since this configuration gives the best performance for all data sets. For each data set, we conducted

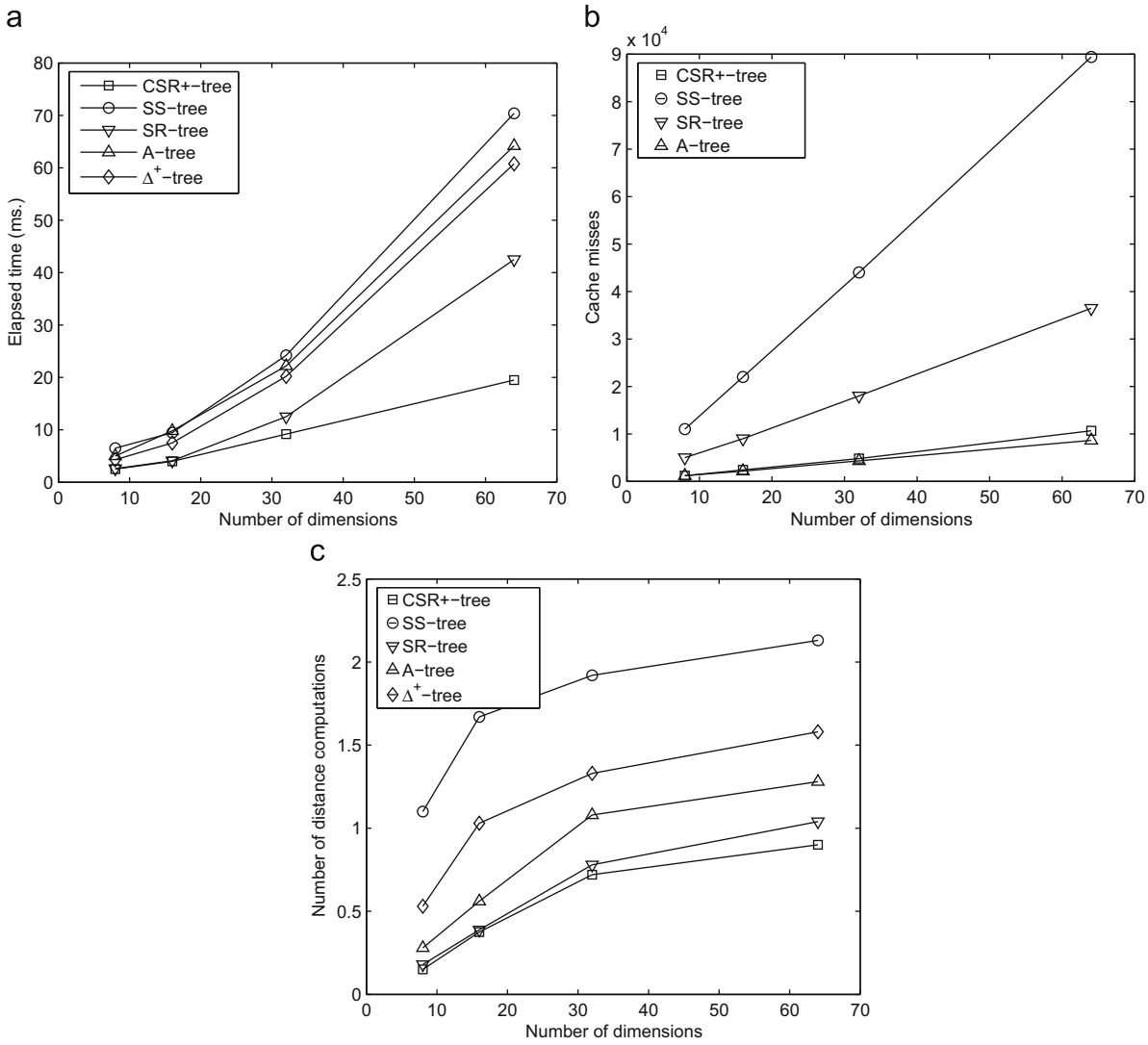



**Fig. 10.** Effect of dimensionality (COREL64, $K$=10): (a) elapsed time; (b) cache misses; (c) distance computations.

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

| | Min | Max | Mean | Std. deviation |
|---|---|---|---|---|
| CSR$^+$-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 |

two experiments. One is varying dimensionality with 10NN queries; another is varying the number of nearest neighbors. Because all four index structures were implemented as main memory indexes, we make the following assumptions: (a) main memory is very small and the data set is very large so that no index structures can reside in main memory; (b) all page accesses are physical transfers. We report here the results on the COREL64 data set (Fig. 13). Similar results were observed on the CENSUS139 data set.

As shown in Fig. 13(a), for dimensionalities greater than 8, the CSR$^+$-tree and the A-tree are superior to the SR-tree and the SS-tree. The performance of the CSR$^+$-tree is about 20% better than that of the A-tree. Fig. 13(b) shows that the CSR$^+$-tree outperforms all other index structures for all values of K.

## 6. Conclusions

We presented the CSR$^+$-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

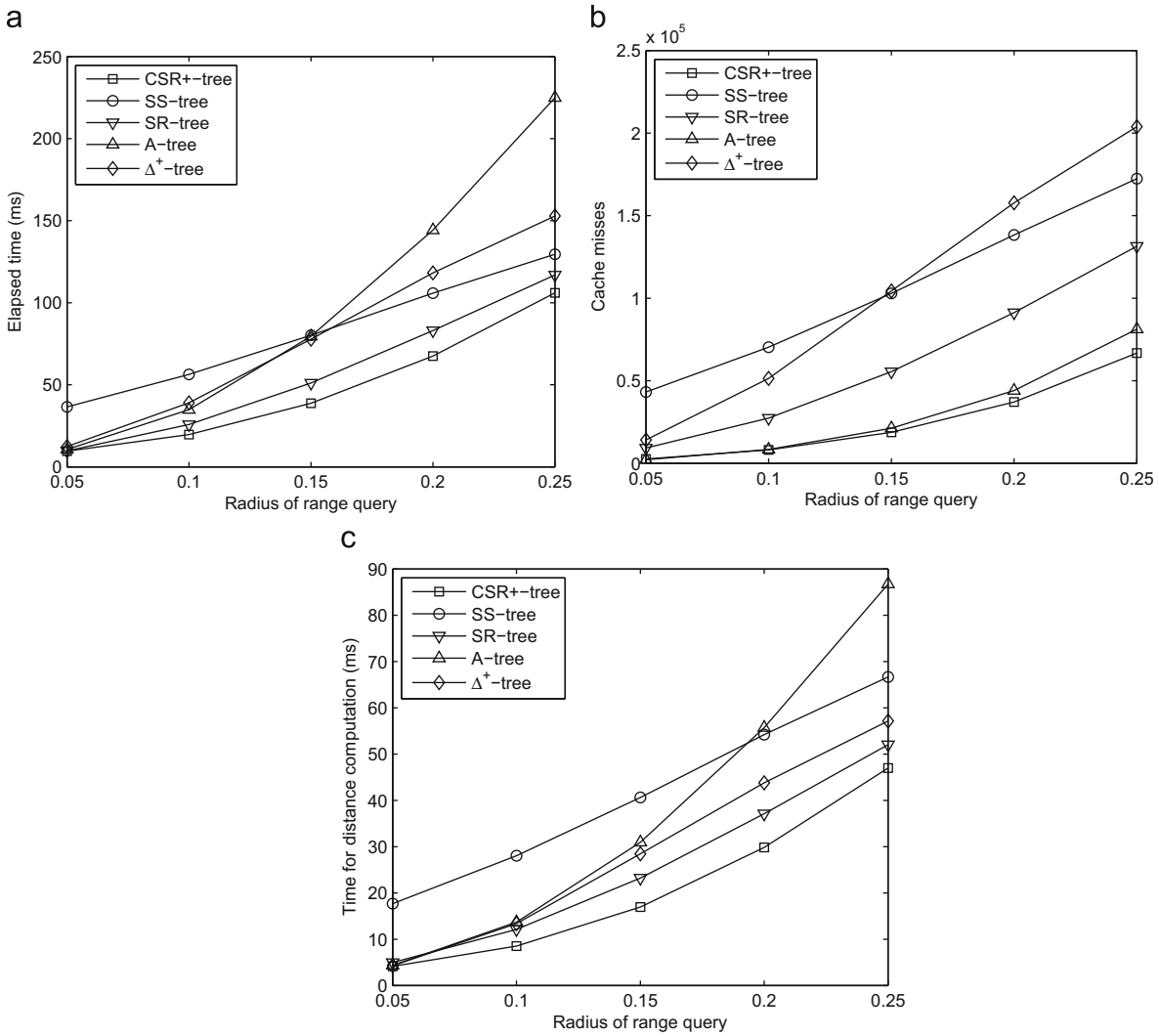



**Fig. 11.** Effect of query radius on range queries (COREL64): (a) elapsed time; (b) cache misses; (c) distance computations.

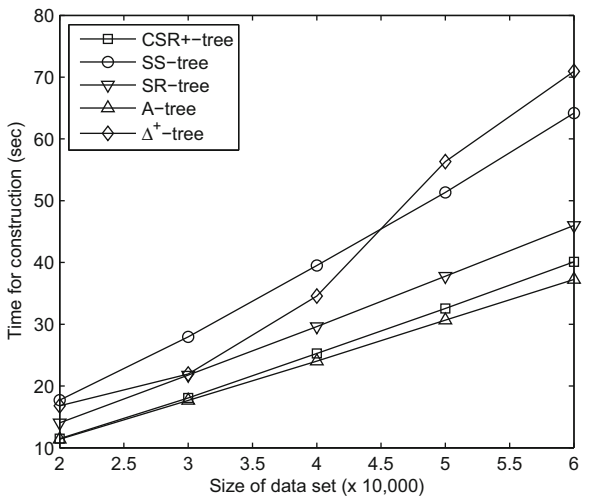



**Fig. 12.** Effect of data size on index construction time (COREL64).

**Table 4**
Index node fan-outs of various index structures (COREL64).

| | CSR$^+$-tree | SS-tree | SR-tree | A-tree | $\Delta^+$−tree |
|---|---|---|---|---|---|
| Leaf node fan-out | 22 | 31 | 32 | 24 | 10 |
| Internal node fan-out | – | 31 | 10 | 12 | 10 |
| QBR node fan-out | 12 | – | – | – | – |
| QBS node fan-out | 22 | – | – | – | – |

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.

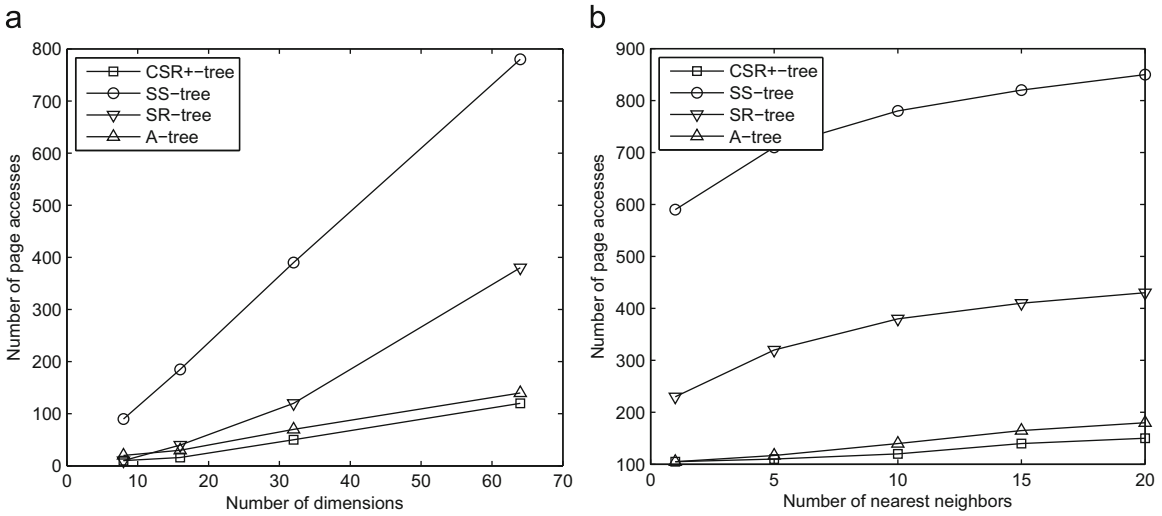



**Fig. 13.** Average number of page accesses (COREL64): (a) varying the number of dimensions; (b) varying the number of nearest neighbors.

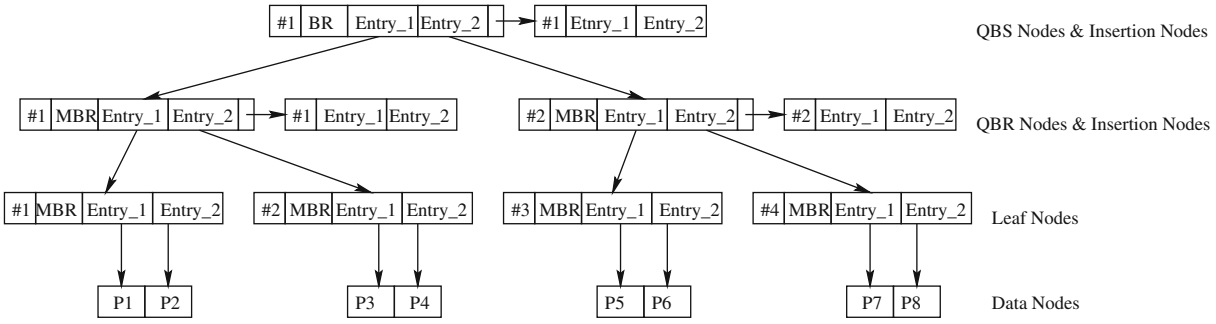



**Fig. 14.** An example of the CSR$^+$-tree.

We conducted extensive experiments to evaluate the CSR$^+$-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$^+$-tree achieves significant improvement over existing index structures, and is the best choice overall.

## Acknowledgments

This work was supported in part by an NSERC Discovery Grant, the Program for New Century Excellent Talents in University (NCET-10-0532), an NSFC Grant (No. 60903108), and the SAICT Experts Program.

## Appendix A. An example of the CSR$^+$-tree

We present here a simple example of the CSR$^+$-tree. Assume that, in a two-dimensional space, there is a set of eight data points, $\{P_1, P_2, \ldots, P_8\}$, where $\vec{p}_i = (i-1, i-1)$. The fan-out of the tree is fixed to be 2. The compression factor is 8 bits. Fig. 14 shows the structure of the CSR$^+$-tree after insertion of all the data points.

There are four leaf nodes in this tree. Table 5 shows the entries in the leaf nodes. Each entry contains $\vec{c}$, $\vec{v}$, and $r$ (in that order), where $\vec{c}$ is the centroid of the QBS, $\vec{v}$ is the bit-string of the centroid, $r$ is the radius from the centroid to the data point that is approximated by the QBS. Now, we describe how leaf node 1 ($L_1$) is obtained. The MBR of $L_1$ bounds points $P_1$ and $P_2$ and is represented as $(\vec{a} = (0,0), \vec{a'} = (1,1))$. By substituting $\vec{p_1} = (0,0)$, $\vec{a}$ and $\vec{a'}$ to Eqs. (1) and (2), we can obtain $\vec{c}_1$, $\vec{v}_1$, and $r_1$. Similarly, by substituting $\vec{p}_2 = (1,1)$, $\vec{a}$ and $\vec{a'}$ to Eqs.(1) and (2), we can obtain $\vec{c}_2, \vec{v}_2$ and $r_2$. Other leaf nodes are obtained in the same way. Note that only $\vec{v}$ and $\vec{r}$ are actually stored in the leaf nodes.

There are two QBR nodes and their associated insertion nodes in this tree. Table 5(a) shows the entries in the QBR nodes. Table 5(b) shows the entries of those insertions nodes associated with the two QBR nodes. Each entry in the QBR nodes contains $(\vec{u}, \vec{u'})$, where $\vec{u}$ and $\vec{u'}$ are the bit-strings of QBR's boundary points. The MBR or QBR node 1 ($RN_1$) bounds points $P_1$, $P_2$, $P_3$ and $P_4$, and is represented as $(\vec{a} = (0,0), \vec{a'} = (3,3))$. Thus by substituting $L_1$.MBR which is represented as $(\vec{b} = (0,0), \vec{b'} = (1,1))$ and $RN_1$.MBR to Eqs. (3) and (4), we can obtain $(\vec{u_1}, \vec{u_1'})$. Similarly, by substituting $L_2$.MBR which is represented as $\vec{b} = (2,2), \vec{b'} = (3,3))$ and $RN_1$.MBR to Eqs. (3)

**Table 5**
Entries in the leaf nodes.

| # | MBR | $Entry_1$ | $Entry_2$ |
|---|---|---|---|
| 1 | (0, 0), (1, 1) | (0.002, 0.002), (0, 0), 0.0028 | (0.998, 0.998), (255, 255), 0.0028 |
| 2 | (2, 2), (3, 3) | (2.002, 2.002), (0, 0), 0.0028 | (2.998, 2.998), (255, 255), 0.0028 |
| 3 | (4, 4), (5, 5) | (4.002, 4.002), (0, 0), 0.0028 | (4.998, 4.998), (255, 255), 0.0028 |
| 2 | (6, 6), (7, 7) | (6.002, 6.002), (0, 0), 0.0028 | (6.998, 6.998), (255, 255), 0.0028 |

**Table 6**
QBR node entries and insertion node entries.

| # | MBR | $Entry_1$ | $Entry_2$ |
|---|---|---|---|
| (*a*) *Entries in the QBR nodes* | | | |
| 1 | (0, 0), (3, 3) | (0, 0), (85, 85) | (170, 170), (255, 255) |
| 2 | (4, 4), (7, 7) | (0, 0), (85, 85) | (170, 170), (255, 255) |

| # | $Entry_1$ | $Entry_2$ |
|---|---|---|
| (*b*) *Entries in the insertion nodes.* | | |
| 1 | (0.5, 0.5), 2 | (2.5, 2.5), 2 |
| 2 | (4.5, 4.5), 2 | (6.5, 6.5), 2 |

**Table 7**
QBS node entries and insertion node entries.

| # | MBR | $Entry_1$ | $Entry_2$ |
|---|---|---|---|
| (*a*) *Entries in the QBS node* | | | |
| 1 | (1.5, 1.5), (5.5, 5.5) | (1.5078, 1.5078), (0, 0), 2.1324 | (5.4922, 5.4922), (255, 255), 2.1324 |

| # | $Entry_1$ | $Entry_2$ |
|---|---|---|
| (*b*) *Entries in the insertion node* | | |
| 1 | (1.5, 1.5), 4 | (5.5, 5.5), 4 |

and (4) we can obtain $(\vec{u}_2,\vec{u}_2')$. Each entry of those insertion nodes contains $P_{centroid}$ and $\omega$, where $P_{centroid}$ is the centroid of all data points contained in the child leaf node, $\omega$ is the number of data points contained in the child leaf node. For example, the entry 1 of $RN_1$ references to the leaf node $L_1$. So, $P_{centroid}$ of the insertion node ($IN_{RN_1}$) is the centroid of all data points contained in the sub-tree rooted at $L_1$, and $\omega_1$ is the number of data points contained in the sub-tree rooted at $L_1$. Note that the two insertion nodes are only used for insertion.

There is one QBS node and its associated insertion node in this tree. Table 6(b) shows the entries in the insertion node. Each entry of the insertion node contains $P_{centriod}$ and $\omega$. $P_{centroid}$ of the insertion node is the centroid of all child centroids contained in $IN_{RN_1}$. $\omega_1$ is the number of data points contained in the subtree rooted at $RN_1$. Note that the insertion node is only used for insertion. Table 6(a) shows the entries in the QBS node, which is also the root of the tree. Each entry of the QBS node contains $\vec{c},\vec{v}$, and $r$, where $\vec{c}$ is the centroid of the QBS, $\vec{v}$ is the bit-string of centroid, $r$ is the radius of the QBS. The BR of the QBS node bounds $P_{centroid_1}$ and $P_{centroid_2}$ that are stored in the insertion node. So the BR is represented as $R$=((1.5, 1.5), (5.5, 5.5)). By substituting $P_{centroid_1}=(1.5,1.5)$ and $R$ to Eqs. (1) and (2), we can obtain $\vec{c}_1,\vec{v}_1$, and $r_1$. Similarly, by substituting $P_{centroid_2}=(5.5,5.5)$ and $R$ to Eqs. (1) and (2), we can obtain $\vec{c}_2,\vec{v}_2$, and $r_2$. Note that only $\vec{v}$ and $r$ are actually stored in the QBS node (Table 7).

## References


[1] S. Berchtold, C. Bohm, H.V. Jagadish, H.-P. Kriegel, J. Sander, Independent quantization: an index compression technique for high–dimensional data spaces, in: ICDE, 2000, pp. 577–588.
[2] S. Berchtold, D.A. Keim, H.-P. Kriegel, The X-tree: an index structure for high-dimensional data, in: VLDB, 1996, pp. 28–39.
[3] S. Berchtold, C. Böhm, H.-P. Kriegel, The pyramid-technique: towards breaking the curse of dimensionality, in: SIGMOD, 1998, pp. 142–153.
[4] T. Bozkaya, M. Ozsoyoglu, Distance-based indexing for high-dimensional metric spaces, in: SIGMOD, 2000, pp. 357–368.
[5] B. Bui, B.C. Ooi, J. Su, K.L. Tan, Indexing high-dimensional data for efficient in-memory similarity search, IEEE Transactions on Knowledge and Data Engineering 17 (3) (2005).
[6] C. Traina Jr., A. Traina, C. Faloutsos, B. Seeger, Fast indexing and visualization of metric data sets using Slim-Trees, IEEE Transactions on Knowledge and Data Engineering 14 (2) (2002) 244–260.
[7] B. Cui, B.C. Ooi, J. Su, K.L. Tan, Contorting high dimensional data for efficient main memory KNN processing, in: SIGMOD, 2003, pp. 479–490.
[8] B. Cui, B.C. Ooi, J.W. Su, K.L. Tan. Main memory indexing: the case for BD-tree, IEEE Transactions on Knowledge and Data Engineering, 2003.
[9] J. Dong, X. Yu, Csr+-tree: Cache-conscious indexing for high-dimensional similarity search, in: Proceedings of the 19th International Conference on Scientific and Statistical Database Management (SSDBM 2007), 2007.
[10] R.F. Santos Filho, A.J.M. Traina, C. Traina Jr., C. Faloutsos, Similarity search without tears: the OMNI family of all-purpose access methods, in: ICDE, 2001, pp. 623–630.
[11] C. Garcia-Arellano, K. Sevcik, Quantization techniques for similarity search in high-dimensional data spaces, in: BNCOD, 2003, pp. 75–94.
[12] G.R. Hjaltason, H. Samet, Ranking in spatial databases, in: SSD, 1995, pp. 83–95.
[13] H.V. Jagadish, B.C. Ooi, K.-L. Tan, C. Yu, R. Zhang, iDistance: an adaptive b+-tree based indexing method for nearest neighbor search, ACM Transactions on Database Systems 30 (2) (2005) 364–397.
[14] I.T. Jolliffe, Principle Component Analysis, Springer-Verlag, New York, 1986.
[15] D.V. Kalashnikov, S. Prabhakar, Fast similarity join for multi-dimensional data, Information Systems 32 (1) (2007) 160–177.
[16] N. Katayama, S. Satoh, The SR-tree: an index structure for high-dimensional nearest neighbour queries, in: SIGMOD, 1997, pp. 369–380.
[17] K. Kim, S.K. Cha, K. Kwon, Optimizing multidimensional index trees for main memory access, in: SIGMOD, 2001, pp. 139–150.
[18] K.-I. Lin, H.V. Jagadish, C. Faloutsos, The TV-tree: an index structure for high-dimensional data, VLDB Journal 3 (4) (1994) 517–542.
[19] H. Murase, S. Shree K, Visual learning and recognition of 3-d objects from appearance, International Journal of Computer Vision 14 (1) (1995) 5–24.
[20] A. Pentland, B. Moghaddam, T. Starner, View-based and modular eigenspaces for face recognition, in: CVPR, 1994, pp. 84–91.
[21] J. Rao, K.A. Ross, Making $B^+$-tree cache conscious in main memory, in: SIGMOD, 2000, pp. 475–486.
[22] J. Rao, K.A. Ross, Cache conscious indexing for decision-support in main memory, in: VLDB, 1999, pp. 78–89.
[23] Y. Sakurai, M. Yoshikawa, S. Uemura, H. Kojima, The A-tree: an index structure for high-dimensional spaces using relative approximation, in: VLDB, 2000, pp. 516–526.
[24] Y. Tao, K. Yi, C. Sheng, P. Kalnis, Quality and efficiency in high dimensional nearest neighbor search, in: SIGMOD, 2009, pp. 563–576.
[25] R. Weber, H.-J. Schek, S. Blott, A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces, in: VLDB, 1998, pp. 194–205.
[26] D.A. White, R. Jain, Similarity indexing with the SS-tree, in: ICDE, 1996, pp. 516–523.
[27] C. Yu, B.C. Ooi, K.-L. Tan, H.V. Jagadish, Indexing the distance: an efficient method to KNN processing, VLDB Journal, 2001, pp. 421–430.