The VLDB Journal (2001) 9: 344–361 / Digital Object Identifier (DOI) 10.1007/s007780100036

Indexing very high-dimensional sparse and quasi-sparse vectors for similarity searches

Changzhou Wang[1,*], **X. Sean Wang**[2,**]

[1] Mathematics and Computing Technology, Phantom Works, The Boeing Company, Bellevue, Washington, USA; E-mail: changzhou.wang@boeing.com
[2] Department of Information and Software Engineering, George Mason University, Fairfax, Virginia, USA; E-mail: xywang@gmu.edu

Edited by Y. Ioannidis. Received: 3 May 1999 / Accepted: 23 October 2000
Published online: 27 April 2001 –

Abstract. Similarity queries on complex objects are usually translated into searches among their feature vectors. This paper studies indexing techniques for very high-dimensional (e.g., in hundreds) vectors that are sparse or quasi-sparse, i.e., vectors *each* having only a small number (e.g., ten) of non-zero or significant values. Based on the R-tree, the paper introduces the xS-tree that uses lossy compression of bounding regions to guarantee a reasonable minimum fan-out within the allocated storage space for each node. In addition, the paper studies the performance and scalability of the xS-tree via experiments.

Key words: Similarity search – High-dimensional indexing structure – Sparse vector – Quasi-sparse vector – Lossy compression

1 Introduction

Similarity queries on complex objects arise naturally in many application domains, such as information retrieval, time series analysis and, in general, digital libraries. Objects are often represented by their feature vectors via manual and/or automatic extraction, and usually a distance function among the vectors is chosen as the similarity measure of the objects.

– In information retrieval, each document is represented by a vector recording its relevance to a given set of terms (i.e., keywords) where the dimension of the vector is the size of the term set. Two documents are considered similar if the inner product of their (normalized) vectors is large enough [20,15].
– Time series, such as stock prices, can be viewed as vectors. Euclidean distance is usually used to measure the similarity among them. Sometimes, transforms, such as the discrete Fourier transform or discrete cosine transform, are applied and the results are used as feature vectors [1,11,23].

The dimension of feature vectors can be very high. For example, in the experiment described in [15], the term set has 1,299 items. However, the features of each individual object are frequently concentrated in a small number of dimensions. As observed in [1,15], in most documents only a few terms occur, and the Fourier transforms of stock prices often have their energy concentrated in the first few frequencies. In other words, the high-dimensional feature vectors in these examples are sparse or quasi-sparse, i.e., non-zero or significant values appear only in a small number of dimensions. Note that, however, significant values in different vectors may appear in different dimensions.

When the number of vectors is very large, some kind of indexing is useful to ensure fast responses. By treating each vector as a point[1] in the high-dimensional space, spatial access methods (SAMs) may be used to index these vectors. However, for very high-dimensional vector sets in which significant values for different vectors appear in different dimensions, a direct application of any SAM (known to us) will have performance problems as discussed in Sect. 3.

In this paper, we introduce the xS-tree, adapted from the R-tree, to index a general category of very high-dimensional (at least in hundreds) vector sets. In this category, the number of significant values for each vector is small (at most in tens) while the significant values of all the vectors in the set taken together may scatter among a large number (at least in hundreds) of dimensions.

Unlike the R-tree which uses rectangles as bounding regions, the xS-tree uses xSquares. An xSquare is a cross product of high-dimensional squares. In order to guarantee a reasonable minimum fan-out (and a constant node size) for the xS-tree, lossy compression is applied to xSquares. In many situations, thanks to the quasi-sparsity of the feature vectors,

The work was partially supported by NSF under the career award 9875114.

* This work was done at George Mason University.

** *Correspondence to:* X. Sean Wang, Department of Information and Software Engineering, Mail Stop 4A4, 4400 University Drive, George Mason University, Fairfax VA 22030-4444, Tel.: +1-703-9931662, Fax: +1-703-9931638

[1] In the following, "point" and "vector" will be used interchangeably.

minimum bounding regions can be approximated by some xSquares which, in turn, can be represented within the storage limit required by the minimum fan-out without introducing intolerable errors.

When the distance measure is fixed as one kind of L_p norm, the xS-tree uses an improved clustering principle. Roughly speaking, this clustering principle favors "squarish" bounding regions (i.e., bounding regions with similar extensions in different dimensions), which are more suitable to be approximated by xSquares than general rectangles.

Finally, we characterize the datasets that can be indexed well by the xS-tree, and study the performance and scalability of the xS-tree for similarity searches through experiments. The result of experiments show that the above strategies work well for many high-dimensional yet sparse or quasi-sparse vector sets.

The rest of the paper is organized into seven sections. High-dimensional sparse and quasi-sparse vectors are categorized by their energy distributions in Sect. 2. In Sect. 3, spatial access methods are discussed in terms of their ability to support similarity search of high-dimensional vectors. The details of adapting the R-tree for our purpose are presented in Sect. 4. The data structure and operations of the xS-tree are described in Sect. 5. The performance and scalability of the xS-tree are shown in Sect. 6 through experiments. Our work is related to other researches in Sect. 7. Finally, the paper is concluded and some future research directions are pointed out in Sect. 8.

2 High-dimensional sparse and quasi-sparse vectors

As mentioned in the introduction, complex objects are represented by feature vectors in many applications. Depending on the nature of these objects, the energy[2] distributions of their feature vectors can be extremely skewed or relatively uniform, while the energy distributions of different vectors can be quite similar or rather different. In this section, we characterize vectors and vector sets by their energy distributions.

A vector is said to be *sparse* if its values in most dimensions are zero. A vector is said to be *quasi-sparse* if its values in most dimensions are non-significant. Here, the significance of a value a is taken as (a^2). Formally, a vector $\boldsymbol{v} = a_1 \cdots a_l$ is quasi-sparse if there exists $D \subset \{1, \ldots, l\}$ where $|D| << l$, and $0 < \sigma \leq 1$ where σ is close to 1, such that $\sum_{k \in D}(a_k^2) \geq \sigma \sum_{k=1}^{l}(a_k^2)$. In other words, the vector $\boldsymbol{v}$ has a large (σ) fraction of its energy concentrated in a small number of dimensions (D). Note that when $\sigma = 1$, the values in dimensions $\overline{D} = (\{1, \ldots, l\} - D)$ are zero, and the vector $\boldsymbol{v}$ is actually sparse. As an example, consider the 200-D quasi-sparse vector $500, 2, -11, 2, 2, 80, 80, \underbrace{2, \ldots, 2}_{191}, -11, 2$, let $D = \{1, 6, 7\}$ and $\sigma = 0.98$, we have $\sum_{k \in D}(a_k^2) = 262800 \geq \sigma \sum_{k=1}^{200}(a_k^2) = 261184$. (The symbols used here, as well as most of the others defined and used in this paper, are summarized in Table 1.)

Given a vector $\boldsymbol{v} = a_1, \ldots, a_l$ and two integers $1 \leq i, j \leq l$, the ith dimension is said to be more significant than the jth dimension if $(a_i^2) > (a_j^2)$. For a given number $0 < \sigma \leq 1$, the ith dimension is said to be *σ-significant* if $\sum_{k \in \{j | a_j^2 > a_i^2\}}(a_k^2) < \sigma \sum_{k=1}^{l}(a_k^2)$. (Here, $\sum_{k \in \emptyset}(a_k^2)$ is defined to be zero.) Intuitively, in order to make up σ fraction of the total energy using several most significant dimensions, we have to use the ith dimension or another dimension that is exactly as significant as the ith dimension. Note that here σ indicates the fraction of energy contributed by a dimension, instead of the significance of the dimension itself. If dimension is σ-significant, it is also σ'-significant for all $\sigma' > \sigma$. Take the previous example of the 200-D vector, the sixth dimension (with value 80) is 0.95-significant, since the first dimension is the only dimension more significant than the sixth dimension, but $a_1^2 = 25000 < 0.95 \sum_{k=1}^{200}(a_k^2) = 250631$. Similarly, the seventh dimension (also with value 80) is also 0.95-significant. In addition, the first dimension is σ-significant for any $0 < \sigma \leq 1$.

We measure the degree of sparsity and quasi-sparsity of a vector with the number of significant dimensions. For a l-D vector $\boldsymbol{v}$, we use $\Theta_{\boldsymbol{v}}(\sigma)$ to denote the set of all its σ-significant dimensions, and use the value $\frac{|\Theta_{\boldsymbol{v}}(\sigma)|}{l}$ to indicate the degree of its quasi-sparsity at the significant level of σ. To continue the above example, the quasi-sparsity of the 200-D vector is $\frac{|\{1,6,7\}|}{200} = 1.5\%$ at the significance level of 0.95. In a special case when $\sigma = 1$, $\frac{|\Theta_{\boldsymbol{v}}(\sigma)|}{l}$ measures the sparsity of $\boldsymbol{v}$.

Table 1. Symbols

Symbols	Comments
l	The dimension of vectors
n	Number of data vectors in a vector set
$D, D_1, \ldots$	A set of dimensions
$\boldsymbol{v}$,$\boldsymbol{v}_1$,$\boldsymbol{v}_2$	(Data) vectors
$\boldsymbol{q}$	Query vectors
V	A set of vectors
σ	Significance level
$\Theta_{\boldsymbol{v}}(\sigma)$	The set of all σ-significant dimensions of $\boldsymbol{v}$
$\frac{\lvert\Theta_{\boldsymbol{v}}(\sigma)\rvert}{l}$	The quasi-sparsity of $\boldsymbol{v}$ at σ-significant level
$\Theta_V(\sigma)$	The set of σ-significant dimensions of all vectors in V
$\frac{\lvert\Theta_V(\sigma)\rvert}{l}$	Overall quasi-sparsity of the vector set V at σ-significant level
L_1, L_2, L_p, L_∞	Vector norm
d	distance
R, R'	Bounding region
$IZ(R, d)$	Influence zone of R with respect to d
X, X'	xSquares
S, S_x, S_y, S_1	Squares in xSquares
M	Storage limit for an xSquare
$c, c_1, \ldots$	Center of xSquares in a single dimension
$r, r_1, \ldots$	Half of the side length of xSquares in a single dimension
$\tau_{\boldsymbol{v}}(k)$	The maximum distance from the k most similar vectors to $\boldsymbol{v}$
$\tau_{\overline{V}}(k)$	The average of $\tau_{\boldsymbol{v}}(k)$ for all $\boldsymbol{v} \in V$.

[2] Energy of a (part of a) vector is calculated by summing up the squares of the values in the (part of the) vector.

For a set of vectors $V = \{\boldsymbol{v}_1, \ldots, \boldsymbol{v}_n\}$, the σ-significant dimensions of V, denoted $\Theta_V(\sigma)$, is $\bigcup_{\boldsymbol{v}\in V} \Theta_{\boldsymbol{v}}(\sigma)$. In other words, if a dimension is (σ-)significant for one vector in the set, it is said to be (σ-)significant *for the vector set*. Similarly, the value $\frac{|\Theta_V(\sigma)|}{l}$, where l is the dimension of vectors, is used to indicate the degree of overall quasi-sparsity of *the set* V at the significant level of σ.

Among all the high-dimensional quasi-sparse vector sets, we distinguish between the vector sets with *similar energy distributions* (i.e., those consisting of vectors having similar energy distributions) and those with *dissimilar energy distributions* (i.e., those consisting of vectors having dissimilar energy distributions) at a given significant level of σ_0. For a vector set V with similar energy distributions, the value $\frac{|\Theta_{\boldsymbol{v}}(\sigma_0)|}{l}$ is close to $\frac{|\Theta_V(\sigma_0)|}{l}$ for each $\boldsymbol{v}$ in V. For example, as shown in [1], the discrete Fourier transforms of stock prices all have their energies concentrated on the first few frequencies, and thus have similar energy distributions.

For a vector set V with dissimilar energy distributions, $\frac{|\Theta_{\boldsymbol{v}}(\sigma_0)|}{l}$ is much smaller than $\frac{|\Theta_V(\sigma_0)|}{l}$ for any $\boldsymbol{v} \in V$. As shown in [15], different documents may use different terms, and the term vectors for a variety of documents are usually a set of sparse high-dimensional vectors with dissimilar energy distributions. In addition, vector sets of this type are quite common due to the wide application of the wavelet transforms, e.g., on images [16,28]. Unlike Fourier transform, wavelet transforms can capture both global and local features of objects. Usually, for each object, the number of *strong* global or local features is quite small; while different objects have different strong (especially local) features. Thus the feature vectors obtained via wavelet transforms are likely to fall into this category.

The similarity between two vectors is often measured by some L_p norm. For a given positive integer p and two vectors $\boldsymbol{v}_1 = a_1 \cdots a_l$ and $\boldsymbol{v}_2 = b_1 \cdots b_l$, the L_p norm of $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ is defined as $L_p(\boldsymbol{v}_1, \boldsymbol{v}_2) = (\sum_{i=1}^{l}(|a_i - b_i|)^p)^{\frac{1}{p}}$. As a special case, L_∞ is defined as $L_\infty(\boldsymbol{v}_1, \boldsymbol{v}_2) = \max\{(|a_i - b_i|)|i = 1 \ldots l\}$.

In information retrieval, term vectors are usually normalized and inner product is used to measure the similarity among them [15]. For a given vector $\boldsymbol{v} = a_1 \cdots a_l$, its normalized vector is $\frac{a_1}{||\boldsymbol{v}||} \cdots \frac{a_l}{||\boldsymbol{v}||}$, where $||\boldsymbol{v}||$ is defined as $\sqrt{\sum_{i=1}^{l}(a_i^2)}$. It is clear that normalization will not change the quasi-sparsity, since for any $0 < \sigma \leq 1$, $\Theta_{\boldsymbol{v}}(\sigma) = \Theta_{\boldsymbol{v}'}(\sigma)$. Given $\boldsymbol{v}_1 = a_1 \cdots a_l$ and $\boldsymbol{v}_2 = b_1 \cdots b_l$, their inner product is defined as $\langle \boldsymbol{v}_1, \boldsymbol{v}_2 \rangle = \sum_{i=1}^{l}(a_i \times b_i)$. Note that two vectors are usually considered more similar to each other when their L_p norm is smaller or their inner product is greater. However, these two measures are closely related to each other. Indeed, for two *normalized* vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, we have $L_2(\boldsymbol{v}_1, \boldsymbol{v}_2) = \sqrt{2 - 2\langle \boldsymbol{v}_1, \boldsymbol{v}_2 \rangle}$.

3 Spatial access methods

By treating vectors as points, we may use spatial access methods (SAMs) to facilitate similarity searches of vectors. However, for *high-dimensional quasi-sparse vector sets with dissimilar energy distributions*, a direct application of these methods will have some performance problems as will be described in this section.

A large number of SAMs have been proposed in the literature (see [12] for a recent survey of SAMs). Many of them work by recursively dividing the potential data region and capturing this process in a tree. Specifically, a bounding region is associated with each node of the tree so that all the regions and data points associated with the sub-tree are contained in this bounding region.

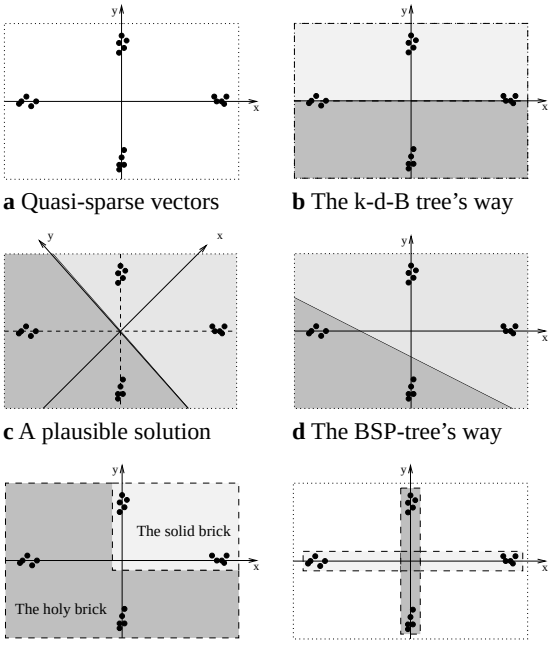
In SAMs such as the k-d-B tree, the LSD tree, and their variants [12,15], the complete information about the bounding region associated with one node is usually stored in multiple places, and the node size is independent of the number of dimensions of the data points. For example, in the simplest variant of the k-d-B tree, the bounding region of each node is divided into two by a hyperplane vertical to an axis, then the two parts become the bounding regions of its child nodes. (To achieve a balanced structure, the two parts divided by the hyperplane should contain similar numbers of points.) Hence, each node only needs to remember the hyperplane used in the division. A complete description of the bounding region can be obtained by the bounding region associated with its parent and the dividing hyperplane.

However, the idea of using only one hyperplane vertical to an axis will sometimes result in poor clustering strategies, especially for very high-dimensional quasi-sparse vectors with dissimilar energy distributions. For example, consider a set of 2-D quasi-sparse vectors (i.e., one value in each vector is far from zero while the other is close to zero) with dissimilar energy distributions, as shown in Fig. 1a, where vectors are mapped into points. A line (vertical to an axis) used to divide them into two groups with roughly the same number of points will be either of the two axes themselves (see Fig. 1b). However, for two of the four point sets that are geometrically clustered together around the axes, the line divides each of them into two groups. At the next level of the tree, the other axis will be selected as the splitting line. In the very high-dimensional case, most of the upper levels will behave in a similar way. It is easily seen that the resulting bounding regions split lots of points that are geometrically clustered.

A plausible solution to this problem is to rotate the coordinate system in a predefined way (see Fig. 1c). However, it is an open research question whether and how this rotation idea can be generalized to a high-dimensional space. Note that the rotation shown in Fig. 1c is different from that achieved through singular value decomposition (SVD).

This problem persists even for vectors with only positive values, provided that they are high-dimensional quasi-sparse and have dissimilar energy distributions. The fundamental reason for this problem is that the discriminative power of any single dimension is limited. For most dimensions, most vectors have insignificant values in those dimensions, and some of them are geometrically clustered even though they have different (insignificant) values in one of these dimensions. Thus, splitting vectors according to only one particular dimension may result in distributing geometrically clustered vectors into different groups.

The above problem can be avoided in several ways. For example, the BSP-tree [12] can use any hyperplane as a splitting plane; the hB-tree [22] uses holy-brick, i.e., a rectangle with some small rectangular regions extracted, as the shape



a Quasi-sparse vectors **b** The k-d-B tree's way
c A plausible solution **d** The BSP-tree's way
e The hB-tree's way **f** The R-tree's way

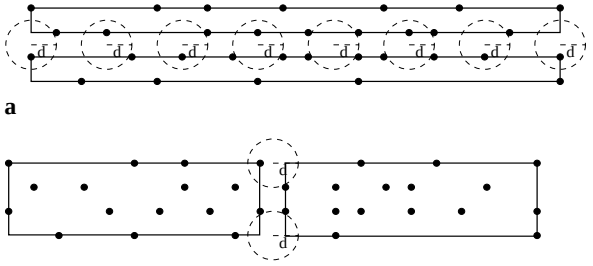
Fig. 1a–f. Different splitting strategies for quasi-sparse vectors

of the bounding region; the R-tree [14] uses closed rectangles as bounding regions but allows bounding regions of the nodes at the same level to overlap. As shown in Fig. 1d–f, no geometrically clustered sets of points are divided into different groups.

However, in all of these trees, the storage space required (in one node) to represent the bounding regions depends on the number of dimensions, and the minimum number of values (required in one node) is in the order of the number of dimensions. When these trees are used to index very high-dimensional (at least in hundreds) points with a typical node size (around 4,096–8,192 bytes), the number of bounding regions that can be stored in one node will be very small. Hence the fan-out will be very low, and the performance of the tree will severely degenerate.

Therefore, in order to index high-dimensional quasi-sparse vector sets with dissimilar energy distributions, it is sufficient to solve either the clustering problem shown in Fig. 1b, or the storage problem shown in Fig. 1d–f. In this paper, we attack the storage problem by compressing representations of bounding regions. In particular, we choose the R-tree as our starting point because: (1) The R-tree is simple and easy to implement and thus robust for practice; and (2) the R-tree uses more compact bounding regions for data points. As shown in Fig. 1f, most irrelevant regions are pruned out. This will potentially improve the performance for similarity searches.

Extensions of other tree structures, such as the hB-tree, are possible but not pursued in this paper. Furthermore, there are a number of variations of the R-tree employing different techniques [2,17,4,26], such as the forced split method used in the R*-tree, to improve the performance. We believe that most of them can be incorporated into the xS-tree. However, for simplicity of presentation, we do not pursue this direction either.



a
b

Fig. 2a,b. Different ways of clustering

4 Indexing high-dimensional quasi-sparse vectors

4.1 Clustering principle

A clustering principle is a rule used to decide how the points are distributed into multiple groups. Each tree structured indexing scheme uses certain clustering principles. A good clustering principle can significantly reduce the average number of nodes visited during the search process. In the R-tree, the clustering principle is to use bounding regions with the minimum total volume. This principle is designed for indexing and query of arbitrary shapes. For similarity searches of points, however, this clustering principle can be improved when the distance measure is fixed as some L_p norm and the query points are uniformly distributed among some (bounded) feasible data region.

As an example, compare the two different ways of clustering in Fig. 2. Obviously, the total volume of the bounding rectangles (drawn with solid lines) in part a is smaller than that in part b. However, for near neighbor searches with Euclidean (L_2 norm) distances d as shown in Fig. 2, the clustering method in part b is better than that in part a. The reason is that a cluster needs to be searched whenever the minimum distance from its bounding rectangle to the query point is less than d. In other words, a cluster needs to be searched if its bounding rectangle intersects with the circle which is centered at the query point and has the radius of d. If the query points are uniformly distributed in a bounded region containing all the data points, i.e., the circle can be anywhere in that region, the chance that both clusters need to be searched is greater in part a than in part b.

The circles in the above example are called *query regions*. Generally, given a vector $\boldsymbol{v} = a_1 \cdots a_l$ and a distance d in terms of some L_p norm, we call the l-D region $\{\mathbf{x} = x_1 \cdots x_l |$ the distance from $\boldsymbol{v}$ to $\mathbf{x}$ is not greater than $d\}$ the *query region*. Figure 3a–c shows the query regions of three different measures for two dimensional vectors.

The situation in Fig. 2 can now be formally explained by considering the interaction of the query region and the bounding region. Given a (bounding) region R, we define its *influence zone*[3] with respect to a distance d in terms of some L_p norm, denoted $IZ(R, d)$, to be the region

$$\{\boldsymbol{v} = v_1 \cdots v_l | \text{ The query region of } \boldsymbol{v} \text{ with the distance } d \text{ intersects with } R\}.$$

[3] Some researchers use "Minkowski sum" to refer to similar concepts. We prefer to introduce a new term since "Minkowski sum" is defined in Mathematics as a more general concept.

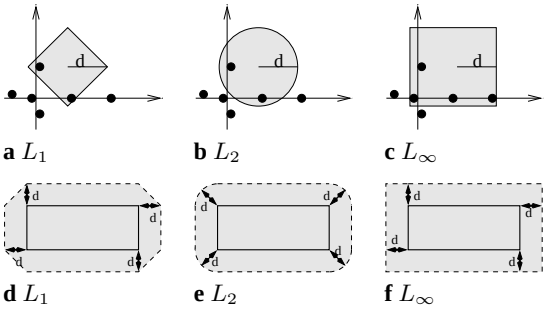


Fig. 3a–f. Query regions and influence zones in 2-D space

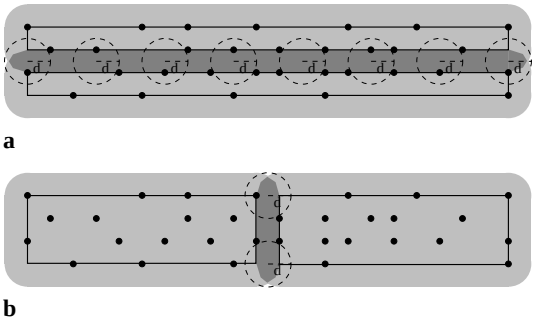


Fig. 4a,b. Influence zones of the two clusterings

In other words, for near neighbor searches with distance d, if a query point falls into $IZ(R,d)$, R needs to be searched. Clearly, the influence zone $IZ(R,d)$ is R itself plus a belt of width d around R. The shapes of the belt at corners are different for different distance measures. Figure 3d–f gives some examples of the influence zones in the 2-D space.

For the above example shown in Fig. 2, the total volume of influence zones in part a is greater than that in part b, as shown in Fig. 4 (the dark shadow is where the two influence zones overlap, and the volume of this overlapped area shall be counted twice for the total volume of the two influence zones). In other words, when query points are uniformly distributed in a bounded feasible region on the plane, the average number of nodes visited in part a for near neighbor searches with distance d is greater than that in part b. Thus, the clustering method in part b is better than that in part a.

For a high-dimensional rectangle, the volume of its influence zone can be calculated in terms of the side lengths of the rectangle and the volume of query regions in lower-dimensional spaces.

Theorem 1. *For a given l-D rectangle $R = \{\boldsymbol{x} = x_1 \cdots x_l | \forall 1 \le i \le l\ (0 \le x_i \le a_i)\}$, and a distance d in terms of some L_p norm, the volume of its influence zone $IZ(R,d)$ is $\sum_{i=0}^{l}(\boldsymbol{R}_i \boldsymbol{Q}_i)$, where $\boldsymbol{R}_i = \sum_{\forall 1 \le k_1 < k_2 < \cdots < k_i \le l} \prod_{1 \le j \le l \wedge j \notin \{k_1,\ldots,k_i\}} a_j$, and $\boldsymbol{Q}_i$ is the volume of the i-D query region with distance d. As special cases, $\boldsymbol{R}_l$ and $\boldsymbol{Q}_0$ are defined as* 1.

Intuitively, the volume of the influence zone is divided into $l+1$ parts. The ith part is the production of the volume of the i-D query region and the total volume of all $(l-i)$-D projections of the rectangle. As special cases, the first part is the volume of the rectangle, and the last part is the volume of the l-D query region. Figure 5 illustrates this for a 2-D rectangle. The proof of this theorem is given in Appendix A.1.

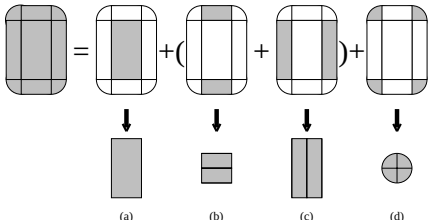


Fig. 5. The volume of the influence zone of a 2-D rectangle

Under the assumption that query points may uniformly appear within a bounded feasible region, the clustering principle used in the xS-tree is to *minimize the total volume of the influence zones of bounding regions*. However, near neighbor searches with different threshold distances may be requested once the indexing structure is built. In addition, for the nearest neighbor search, the threshold distance is not given at all. Although this makes the accurate calculation of influence zone infeasible during the process of building the xS-tree, the accuracy of the value of the distance used in the calculation will not affect the correctness of the xS-tree. In practice, system designers can pick up a typical value according to the application requirement. Interestingly, the evaluation time for queries with distance values greater than the chosen one is guaranteed to be less when the influence zone is considered (versus only the volume of bounding region is considered). Hence, the overall performance is likely to improve when the value is selected appropriately. In addition, the distance value used to calculate the influence zone can be extended to a distribution function, which takes into account the probability of queries using different distances. In this case, the volume of the influence zone will be a distribution function. This extension is not pursued in this paper for simplicity of presentation.

Using the volume of the influence zone as the criterion in choosing bounding regions tends to favor squarish regions, i.e., rectangular regions with similar extensions in different dimensions. Formally,

Theorem 2. *Let R be an l-D ($l \ge 2$) rectangle with side lengths $a_1, \ldots, a_l$. Let R' be another l-D rectangle with side lengths $a_1', \ldots, a_l'$. For any distance d in terms of some L_p norm, the volume of $IZ(R,d)$ is greater than that of $IZ(R',d)$ if there exists $1 \le m_1 \ne m_2 \le l$ such that (1) $a_i' = a_i$ for $1 \le i \le l$, $i \ne m_1$ and $i \ne m_2$, (2) $a_{m_1} a_{m_2} = a'_{m_1} a'_{m_2}$, and (3) $|a_{m_1} - a_{m_2}| > |a'_{m_1} - a'_{m_2}|$.*

The proof is given in Appendix A.2.

4.2 Bounding regions

As discussed in Sect. 3, in order to adapt the R-tree to index high-dimensional quasi-sparse vector sets with dissimilar energy distributions, we compress the representations of bounding regions to meet the storage requirement. The compression procedure involves two steps: (1) approximate the bounding region by enlarging it; and (2) (losslessly) compress the representation of the approximated region. The first step is to change the shape of the region into some desirable form such that

the representation of the resulting region can be compressed more effectively in the second step. The error introduced by this compression procedure is measured by the increase in the volume of the influence zone of the bounding region. To achieve good performance, this error should be small, while the compressed representation of the approximate region can be stored within the storage limit.

Clearly, the compression procedure and the error introduced by the procedure depends on the specific shape of bounding regions and the representation methods used. In this paper, we propose to use xSquares as bounding regions. Intuitively, "xSquare" is another name for rectangle but focuses on the equal size lengths in some dimensions. Specifically, an l-D *xSquare* is a special rectangle defined as follows: Let $D_1, \ldots, D_k$ be a partition of all l dimensions (i.e., $\bigcup_{i=1}^{k} D_i = \{1, \ldots, l\}$, and $D_i \cap D_j = \emptyset$ for any $1 \leq i \neq j \leq l$). Given values $c_1, \ldots, c_l$ and $r_1, \ldots, r_k$, the point set $\{\mathbf{x} = x_1 \cdots x_l | \forall 1 \leq i \leq k (\forall j \in D_i (c_j - r_i \leq x_j \leq c_j + r_i))\}$ is an xSquare, and $\mathbf{c} = c_1 \cdots c_l$ is the center point. For $i = 1, \ldots, k$, the projection of the xSquare on D_i is a square with side length $2r_i$. Clearly, an xSquare can be viewed as a cross product of some low-dimensional squares (and hence the name).

In general, xSquares are appropriate for bounding high-dimensional quasi-sparse vectors. For a minimum bounding rectangle of a set of similar high-dimensional quasi-sparse vectors, the side lengths of the rectangle in insignificant dimensions (for all vectors bounded in this rectangle) will be similar to each other and thus can be approximated by a square in these dimensions without introducing large errors. As the significant values vary from one to the other, side lengths in significant dimensions can be divided into different groups and each group can be approximated by a square. The cross product of these squares naturally gives rise to an xSquare.

The usage of xSquares as bounding regions are especially amenable to the clustering principle proposed in the above section. Recall that the clustering principle favors squarish shapes, thus the bounding regions are likely squarish. Consequently, the error introduced by the approximation tends to be small. Furthermore, the second step of the compression procedure can readily use the fact that many side lengths in an xSquare are the same.

5 The xS-tree

5.1 Data structure

Similar to the R-tree, the xS-tree consists of a hierarchy of nodes, among which we distinguish between leaf nodes and non-leaf nodes. A leaf node of the xS-tree contains a set of items, which could be either vectors or pointers pointing to vectors. There is an upper bound and a lower bound for the number of items in one leaf node. Usually the lower bound is one half of the upper bound. Generally, once a leaf node is reached during the search, each vector within the node needs to be checked to see if it satisfies the search criteria. When the cost of computing distances between high-dimensional vectors is high, the upper bound shall be small so that the search process on the xS-tree can narrow down to only a few vectors. On the other hand, the lower bound shall be large (yet it must be smaller than the upper bound) in order to ensure the minimum storage utilization and thus limit the maximum level of the tree for a given set of vectors. The trade-off between these two requirements shall be made to minimize the average search cost. For simplicity, we do not pursue this direction.

A non-leaf node contains a set of branches. Each branch is a pair consisting of a pointer pointing to a child node and an xSquare which is the approximation of the *minimum bounding region* of the child node. Similar to the leaf nodes, there is an upper bound and a lower bound for the number of branches in each non-leaf node. Furthermore, in order to keep a constant node size, a limited (constant-size) storage space is allocated to compactly store each xSquare (of a branch).

To facilitate compact representation, each xSquare in the xS-tree has a special (high-dimensional) square, which is expected to approximate all insignificant dimensions. Its dimensions are those that do not appear in other squares and hence not explicitly represented, and its center point is required to be on the diagonal, i.e., the values in all dimensions are the same.

For example, the 200-D xSquare with the center values: $500, 2, -11, 2, 2, 80, 80, \underbrace{2, \ldots, 2}_{191}, -11, 2$ and side lengths: $50, 1, 5, 1, 1, 50, 50, \underbrace{1, \ldots, 1}_{191}, 5, 1$ is represented by:

$$\langle\langle\{1\}, 500, \{6, 7\}, 80, 50\rangle\langle\{3, 199\}, -11, 5\rangle, \langle 2, 1\rangle\rangle .$$

This xSquare has two squares in addition to the special one. The first square is in dimensions $\{1, 6, 7\}$, the second square is in dimensions $\{3, 199\}$, and the special square is in dimensions $\{2, 4, 5, 8, 9, \ldots, 197, 198, 200\}$. For the first square, the side length is 50, and the three dimensions are divided into two groups $\{1\}$ and $\{6, 7\}$ since each group has a different center coordinate value, namely, 500 for the first group and 80 for the second group. For the second square, as the center point is on the diagonal, its two dimensions are kept in the same group.

In general, an l-D xSquares with $k + 1$ squares is first represented as

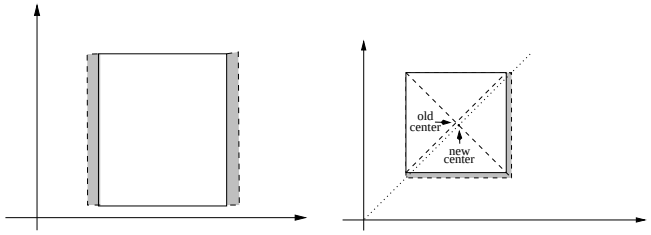
$$\begin{array}{ll}
\langle\langle D_{1,1}, c_{1,1}, \ldots, D_{1,s_1}, c_{1,s_1}, 2r_1\rangle, & \text{(the first square)} \\
\ldots, & \\
\langle D_{i,1}, c_{i,1}, \ldots, D_{i,s_i}, c_{i,s_i}, 2r_i\rangle & \text{(the ith square)} \\
\ldots, & \\
\langle D_{k,1}, c_{k,1}, \ldots, D_{k,s_k}, c_{k,s_k}, 2r_k\rangle & \text{(the kth square)} \\
\langle c_{k+1}, 2r_{k+1}\rangle\rangle, & \text{(the special square)}
\end{array}$$

where $D_{i,j} (1 \leq i \leq k, 1 \leq j \leq s_i)$ is a set of dimensions. Here, the ith square has the side length of $2r_i$, and its dimensions D_i are (further) partitioned into s_i groups: $D_{i,1}, \ldots, D_{i,s_i}$. For each dimension in the jth group $D_{i,j}$, where $1 \leq j \leq s_i$, the coordinate of its center point is $c_{i,j}$. The last line is used to record the center value and the side length of the special square mentioned above.

For each group of dimensions $D_{i,j}$, if it contains more than three consecutive dimensions, a variation of run-length compression is applied. For example, a group of dimensions $\{3, 5, 6, 7, 9\}$ is compactly represented as $\{3, 5\#3, 9\}$.

5.2 xSquare compression

For a given minimum bounding region represented by an xSquare in the xS-tree, the above representation (with the run-



a Merge lower dim. squares **b** Shift a square's center

Fig. 6a,b. Two lossy compression operations

length compression) may not guarantee that the xSquare can fit into the (constant-size) allocated storage space. A lossy compression procedure is invoked to find a good xSquare which encloses the given one and whose (compressed) representation can fit into the allocated storage space.

Lossy compression operations

Two lossy compression operations, namely *merging* and *shifting*, are used in the xS-tree. Figure 6 illustrates them in a 2-D space.

The merging operation merges two squares in two non-overlap sets of dimensions into a higher-dimensional square. More specifically, for two sets of dimensions $D_1 = \{i_1, \ldots, i_{l_1}\}$ and $D_2 = \{j_1, \ldots, j_{l_2}\}$ where $D_1 \cap D_2 = \emptyset$, a square S_1 in D_1 with center point $\mathbf{c} = c_{i_1} \cdots c_{i_{l_1}}$ and side lengths $2r_1$, and a square S_2 in D_2 with center point $\mathbf{c} = c_{j_1} \cdots c_{j_{l_2}}$ and side lengths $2r_2$ are merged into one square S in $D_1 \cup D_2$ with center point $\mathbf{c} = c_{i_1} \cdots c_{i_{l_1}} c_{j_1} \cdots c_{j_{l_2}}$ and side lengths $2r$, where $r = \max(r_1, r_2)$. In Fig. 6a, two 1-D squares (i.e., intervals) are merged into a 2-D square by padding the shadow area. At least one value is saved in the representation scheme discussed above due to the merge.

When the side lengths of the two (lower-dimensional) squares are similar, the merging of them does not cause much of a loss. Actually, the merging operation is especially useful in combining insignificant dimensions into one square.

The shifting operation shifts the center of a square to the diagonal to reduce the storage space required to represent the center point. More specifically, for a set of dimensions $D = i_1, \ldots i_k$ and a square S in D with center point $\mathbf{c} = c_{i_1} \cdots c_{i_k}$ and side lengths $2r$. S can be shifted to S' with the center point $\mathbf{c}' = c' \cdots c'$ and side length $2r'$ such that S is contained in S'. The S' with the minimum volume (and thus the minimum influence zone) is given by $c' = \frac{\min_{i \in D}(c_i) + \max_{i \in D}(c_i)}{2}$, and $r' = r + \frac{\max_{i \in D}(c_i) - \min_{i \in D}(c_i)}{2}$. In Fig. 6b, the 2-D square is shifted by padding the shadow area. Shifting a k-D square will save at least $(k-1)$ values for the representation scheme given above.

We call one instance of applying either of these two compression operations a *compression step*. In other words, a merging of two specific squares is a compression step, and a shifting of a specific square is also a compression step. For a given xSquare, both compression operations can be applied to many different squares in the xSquare, which results in many different choices of compression steps. All the possible compression steps will enlarge the xSquare and its influence zone. *The volume of the influence zone of the xSquare is used as the criterion to choose among different compression steps,* as we will discuss in detail in the following. The one giving the smaller influence zone is preferred.

The xSquare compression algorithm

The compression procedure is to find a series of compression steps so that the result xSquare can be stored in the allocated space while the increased influence zone is small. Since the total search space is exponential in terms of the number of dimensions, an exhaustive search will be computationally costly. A greedy algorithm, shown in Fig. 7, is used for this task instead.

Based on the observation that shifting higher-dimensional squares saves more storage space than shifting lower-dimensional squares or merging two squares, the following heuristic is used in the greedy algorithm: *if the xSquare cannot be compressed to fit into the storage limit by using only a series of merging steps, each merging step on the xSquare is followed directly by a shifting step on the result of the merging (we call this a merging-and-shifting step)*. The intuition behind this heuristic is that for a merging step in a given series of compression steps, if no shifting step after it can use its result, this merging step and other merging steps dependent on it can be deferred to the end of the algorithm.

The algorithm **Compress** in Fig. 7 takes two parameters: an xSquare X and a storage limit M. Since an xSquare can be, at most, compressed into one *square* whose center point is on the diagonal, M must be no less than the storage size required to represent such a square. Note that this square only has two values, one for the center point and one for the side length. The output of the algorithm is an xSquare that encloses X and its representation requires no more storage space than M.

After some preparation done in steps (1) and (2), each square in the xSquare X' has its center point on the diagonal. The algorithm then (step (3)) uses the heuristic mentioned above to compress the xSquare by a series of merging-and-shifting operations in a greedy way. The storage size of X' is calculated as the storage space needed to represent X' as described in Sect. 5.1. The cost of merging or shifting is calculated as the enlarged volume of X''s influence zone. In step (4), the algorithm works in a similar way. However, there are two choices: either to compress X' by using only a series of merging steps (step 4.1), or apply a merging-and-shifting step (step 4.2).

The algorithm **TryMergeOnly** is to find a series of merging-only steps to compress an xSquare to meet the storage limit. When the squares in the xSquare are sorted by their radii, *the best merging-only series that results in the least enlargement of the volume of the xSquare's influence zone only contains steps that merging consecutive squares*. See the discussion in Appendix A.3 for the proof of this property. In Fig. 7, this property is exploited and a greedy algorithm is used.

Compress: Compress an xSquare	
Input	An xSquare X, and a reasonable storage limit M.
Output	An xSquare X' such that X' encloses X, and the storage space needed to represent X' is within M.
Method	(1) For each square S in X (1.1) Break S into the minimum number of squares whose center points are on the diagonal. (2) Create an xSquare X' using all the squares generated in (1.1). (3) While (it is impossible to compress X' into M by only using merging steps, i.e., TryMergeOnly does not work yet) do (3.1) For each pair of squares in X', (S_x, S_y), do Calculate the total cost (in terms of the increased volume of influence zones) of merging S_x and S_y into S_z and shifting S_z; (3.2) Let (S_1, S_2) be the pair with the smallest cost calculated in (3.1); Merge S_1 and S_2 into S and then shift S in X'; (4) While (StorageSizeOf$(X') > M$), do (4.1) Call the algorithm TryMergeOnly with (X', M), let the result be (c_1, Y); (4.2) For each pair of squares (S_x, S_y) in X', do Calculate the total cost of merging S_x and S_y into S_z and shifting S_z; (4.3) Let (S_1, S_2) be the pair with the smallest cost c_2 calculated in (4.2); (4.4) If $(c_1 < c_2)$ let X' be Y; else merge S_1 and S_2 into S and then shift S in X';

TryMergeOnly: Compress an xSquare only by merging	
Input	An xSquare X, and a reasonable storage limit M.
Output	The cost C and an xSquare X' such that: X' encloses X, the storage required to represent X' is within M, and the center point of X' is the same as that of X.
Method	(1) Create an xSquare X' by copying X; (2) Sort squares in X' according to their radii; (3) While (StorageSizeOf$(X') > M$), do (3.1) For each consecutive pair of squares (excluding the special one) in X', Calculate the cost of merging these two squares; (3.2) Let $\langle S_1, S_2 \rangle$ be the consecutive pair with the smallest cost calculated in (3.1) Merge S_1 and S_2 into S, remove S_1 from X', replace S_2 by S; (4) Let C be the total (accumulative) cost in (3.2)

Fig. 7. The xSquare compression algorithm

5.3 Operations

Insertion and deletion

The process of inserting a vector into or deleting a vector from the xS-tree is similar to that of the R-tree except for the following two aspects: (1) The clustering principle of minimizing the total volume of the influence zones of bounding regions is used whenever a bounding region is changed. This happens when a child node is selected during the insertion, or points are redistributed during the splitting process; (2) once a minimum bounding region is calculated, the algorithm **Compress** is invoked to compress it to meet the storage limit. The resulting xSquare will be used as the bounding region from then on.

Specifically, to insert a point into the xS-tree, each xSquare in the root is examined to see how much its influence zone has to be enlarged if the point is inserted into its associated node. The one with *the smallest enlargement* will be identified and the process goes on as if the point is to be inserted into the sub-tree rooted at the child node associated with the identified xSquare. When a leaf node is reached, the point is simply inserted if the leaf still has room for it and all the ancestor bounding xSquares are updated. Otherwise, a split of the leaf node takes place.

To split a leaf node, all points within it need to be divided into two groups, such that the total volume of the influence zone of the bounding xSquares for both groups is as small as possible. (This problem is computationally expensive. The xS-tree uses the "quadratic-cost algorithm" proposed in the R-tree [14] to find a sub-optimal solution.) A new leaf is created and the two groups of points are put into the new leaf and the old leaf. The xSquares are also updated.

The process of deletion is simpler. An exact search process is first invoked to locate the point to be deleted. The point is simply deleted from the leaf node which contains it if the leaf node still satisfies the lower bound requirement. The bounding

xSquares of the ancestor nodes are updated. Otherwise, the leaf node is deleted and all the orphaned points are re-inserted. This may trigger the deletion of the parent of the leaf node, which is treated in a similar way.

Similarity searches

There are two major kinds of similarity searches, namely, near neighbor search and nearest neighbor search. The near neighbor search problem is to find all points that are contained in a query region. To accurately check whether a query region intersects with an xSquare, the minimum distance from the query point to the xSquare is first calculated, and it is then compared with the threshold distance given in the query.

The minimum distance, in terms of some L_p norm, from a point $\mathbf{q} = q_1 \cdots q_l$ to an xSquare $X = \{\mathbf{x} = x_1 \cdots x_l | a_i \leq x_i \leq b_i \text{ for } i = 1, \ldots, l\}$ is calculated as follows: For $i = 1, \ldots, l$, let d_i be the minimum distance from q_i to the interval $[a_i, b_i]$, i.e.,

$$d_i = \begin{cases} q_i - b_i & \text{if } q_i > b_i \\ 0 & \text{if } a_i \leq q_i \leq b_i \\ a_i - q_i & \text{if } q_i < a_i \end{cases}$$

Then the minimum distance from $\mathbf{q}$ to X is $(\sum_{i=1}^{l} (d_i)^p)^{\frac{1}{p}}$ for L_p norm, or $\max\{d_1, \ldots, d_l\}$ for L_∞ norm.

For the k nearest neighbor search problem, we use a divide-and-conquer method as described in [25]. A priority queue is maintained to store all the promising nodes to be checked, and a second queue is used for keeping the candidate nearest points. Beginning at the root node, the method examines the bounding xSquares of all child nodes of the current node. Some of the child nodes are pruned out as described below, while others are put into the first queue in the order of their "priorities". Then the first node in the first queue is removed from the queue and treated as the new "current" node and the process goes on. When a leaf node is reached, all points in it are checked and put into the second queue in the order of their distances from the query point. The kth point in the second (point) priority queue, if available, gives the upper bound of the distance threshold and can be used for the above node pruning process by calculating the minimum distances from their xSquares to the query point. Note that the correctness of the algorithm does not depend on the priority calculation. However, a carefully chosen priority can improve the search performance tremendously.

To calculate the priorities of nodes, two kinds of distances, namely **min** and **minmax** distances from the query point to a bounding region, are used in [25]. The **min** distance is the minimum distance from the query point to the bounding region of the node. The **minmax** distance is "the minimum of the maximum possible distance" [25] from some point in the bounding region to the query point. When the bounding region is the *minimum* bounding rectangle, both distances can be easily calculated, as discussed in [25] and shown in Fig. 8.

However, the xSquares used in the xS-tree are not the minimum bounding regions due to the lossy compression, and the **minmax** distance is not easy to calculate. Instead, the **max** distance is used in this paper. The **max** distance is simply the maximum distance from the query point to the bounding xSquare. Specifically, the **max** distance, in terms

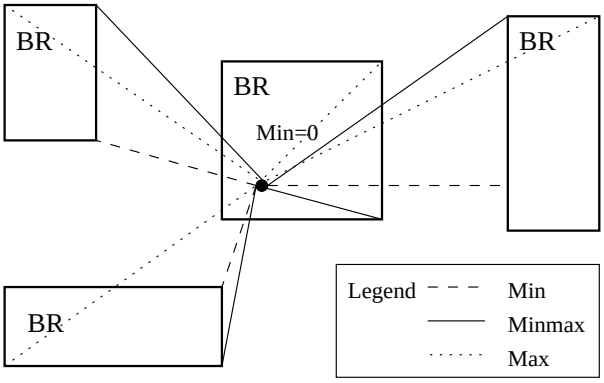


Fig. 8. Three distances used in nearest neighbor searches, based on Fig. 5 in [25]

of some L_p norm, from a point $\mathbf{q} = q_1 \cdots q_l$ to an xSquare $X = \{\mathbf{x} = x_1 \cdots x_l | a_i \leq x_i \leq b_i \text{ for } i = 1, \ldots, l\}$ is $(\sum_{i=1}^{l} (\max(|q_i - a_i|, |q_i - b_i|))^p)^{\frac{1}{p}}$, or $\max\{|q_1 - a_1|, |q_1 - b_1|, \ldots, |q_l - a_l|, |q_l - b_l|\}$ for L_∞ norm.

Clearly the average distance from the query point to points in (the sub-tree of) a node is between **min** and **max**. Hence we use α**min** $+ (1-\alpha)$**max**, for some $0 \leq \alpha \leq 1$, as the priority of a node. When the constant α is chosen such that α**min** $+ (1-\alpha)$**max** is close to the average distance mentioned above, the performance of the xS-tree is expected to be improved. The use of the **max** distance also introduces new opportunities for pruning the xS-tree. For example, if a sub-tree associated with a node has more than k points, then the **max** distance from the query point to the bounding xSquare of this node can also serve as the pruning threshold. Indeed, if the **min** distance of the bounding xSquare of a node to the query point is greater than the above **max**, then the node can be safely dropped without examining its content.

6 Performance

We study the performance of the xS-tree through experiments, and compare the xS-tree with the SR-tree [18]. The SR-tree is chosen since it is one of the latest indexing structures developed for high-dimensional non-uniform vectors, and outperforms other similar indexing structures including the R*-tree, X-tree [4], and SS-tree [29], as reported in [18].

The SR-tree uses L_2 norm as the distance measure and an intersection of a bounding rectangle (as in the R-tree) and a bounding sphere (as in the SS-tree) as the bounding region. The use of bounding spheres "enhances the performance" of similarity search, and the use of bounding rectangles improves the "disjointness" among bounding regions. Thus the SR-tree requires $3l$ values, where l is the dimension of vectors, to record each bounding region. For very high dimension vectors, the SR-tree has to use a vary large node size. For example, when $l = 512$ and the node fan-out is 8, $512 \times 3 \times 8 = 12288$ values ($49,152$ bytes) are necessary for each non-leaf node. In practice, the SR-tree itself may requires storage space comparable to the whole dataset.

We choose to conduct experiments on large synthetic datasets. By generating data ourselves, we can control various characteristics of different datasets, and hence study the per-

formance and the scalability of the xS-tree in a comprehensive way. We plan to conduct in the future some experiments using data collected from some application systems. In the following, we first discuss the algorithm that generates the datasets as well as the characteristics of the datasets in Sect. 6.1, and then report the performance of the xS-tree versus the SR-tree on these datasets in Sect. 6.2.

6.1 Datasets

We use four parameters to generate our synthetic datasets: the dimension of vectors (denoted l), the number of vectors (denoted n), an integer s ($1 \leq s \leq l$) used to control the quasi-sparsity of the vectors, and a real number f ($0 < f \leq 1$) used to control the variation of the energy distributions of the vectors in a set. The process is as follows: (1) generate a seed vector; (2) produce a new vector from the seed by exchanging some significant values with some insignificant ones in the seed vector; (3) slightly change the value of the new vector in each dimension; and (4) use the new vector as the new seed and the continue the process until all the required n vectors are generated. The detail is shown in Fig. 9.

An alternative way is to invoke the seed generation step to produce each vector; thus, there is no dependency between vectors. For such a vector set, as will be shown in Fig. 10, if the dimension is high and the set size is small, the distances are likely to be similar for most pairs of vectors. As a result, for a given query vector, either too many vectors are similar, or (almost) none of them are. However, in reality, people use similarity queries only when they expect to find a reasonable small number of vectors (that are similar to a given one).

In contrast, in the vector generation algorithm, with a properly chosen f, the distance between a seed and the vector generated from it will be much smaller than that of an arbitrary pair of vectors. This property can be characterized in the following way: for a vector $\boldsymbol{v}$ in a given vector set V and some L_p norm distance measure, let $\tau_{\boldsymbol{v}}(k)$ be the maximum distance to the k most similar vectors to $\boldsymbol{v}$ (for $1 \leq k \leq |V|$), and $\tau_{\overline{V}}(k)$ be the average of $\tau_{\boldsymbol{v}}(k)$ for all $\boldsymbol{v} \in V$. Obviously, if $\tau_{\overline{V}}(k)$ does not change much for different k (especially small k's), the result of similarity search will not be very interesting.

The generation algorithm can be varied in many ways. For example, we may start with more than one seed to generate vectors simultaneously, or we can generated more than one vector from each seed. However, such variations usually result in a number of (geometrical) clusters of vectors and only affect a few highest levels of the xS-tree.

In Fig. 10, we show some characteristics of a typical vector set V in our experiment, which is generated with the parameters: $n = 1,000$, $l = 512$, $s = 16$, and $f = \frac{1}{16}$. Figure 10a shows the 501st vector $\boldsymbol{v}_{501}$. Figure 10b shows its quasi-sparsity by presenting the $\frac{|\Theta_{\boldsymbol{v}_{501}}(\sigma)|}{l}$ as a function of σ. Figures 10d and e show the average quasi-sparsity of the $1,000$ vectors, i.e., the average value of $\frac{|\Theta_{\boldsymbol{v}}(\sigma)|}{l}$ for all $\boldsymbol{v} \in V$, and the overall quasi-sparsity of the $1,000$ vectors, i.e., $\frac{|\Theta_V(\sigma)|}{l}$. In Fig. 10f, for each dimension $1 \leq i \leq 512$, the number of vectors whose ith dimension is 0.9-significant is presented. In Fig. 10c, $\tau_{\overline{V}}(k)$ is compared to $\tau_{\overline{V'}}(k)$ where V' is a uniformly distributed vector set generated with parameters $n = 1,000$, $l = 512$, and $s = 16$.

Note that in Fig. 10, on average there are less than 5% (25.6) dimensions that are 0.9-significant for one vector, while, as a set, even 0.75-significant dimensions scatter in more than 50% (256) dimensions. In addition, $\tau_{\overline{V'}}(k)$ does not change much while $\tau_{\overline{V}}(k)$ increases linearly from 0 for $1 \leq k \leq 50$. In other words, on average, the first 50 most similar vectors in V can be distinguished quite well from all other other vectors. We believe that the vector sets we generated mimic real datasets better than the uniformly distributed vectors sets, especially when the similarity queries are applicable and meaningful.

Altogether, we generate four groups of datasets using different parameters. In each group, we vary one parameter to generate 5–8 datasets. We then build the xS-tree and the SR-tree on each of these dataset and test the search performance. The default parameters used to generate the datasets are $n = 1,000$, $l = 512$, $s = 16$, and $f = \frac{1}{16}$. In Fig. 11, we show the overall (quasi-)sparsity ($\frac{|\Theta_V(\sigma)|}{l}$) and the average number of significant dimensions (mean$\{(|\Theta_{\boldsymbol{v}_i}|)|\forall \boldsymbol{v}_i \in V\}$) for all datasets.

In Fig. 11a, the datasets are generated with different dimensions (128, 256, 512, $1,024$, and $2,048$). In Fig. 11b, the datasets are generated with different numbers of vectors (250, 500, 750, $1,000$, $2,000$, $3,000$, $4,000$, and $5,000$). In Fig. 11c, the datasets are generated with different s's (4, 8, 16, 32, and 64). In Fig. 11d, the datasets are generated with different f's ($\frac{1}{64}$, $\frac{1}{32}$, $\frac{1}{16}$, $\frac{1}{8}$, and $\frac{1}{4}$).

Due to the randomness of the vector generation algorithm, the number of significant dimensions does not behave exactly as expected. However, the total number of significant dimensions for each set is quite large (note that the number shown in the figure is the fraction instead of the actual number), while the average number of significant dimensions are quite small except for the dataset with dimension of $2,048$. Furthermore, in Fig. 11c, the fractions of significant dimensions for each datasets are quite similar while the average numbers are different when $\sigma > 0.9$. In contrast, the fractions of significant dimensions for different datasets in Fig. 11d are quite different.

6.2 Experiment on the xS-tree

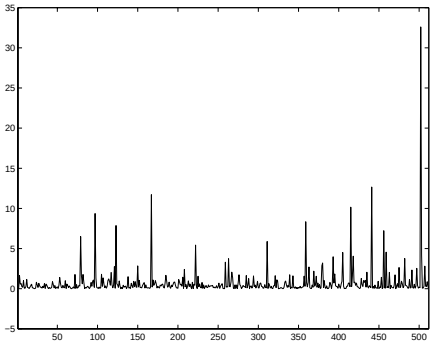
We test the performance of the xS-tree versus the SR-tree on each generated dataset, using the L_2 norm as the distance measure and assuming that the block size is $2,048$ bytes. Specifically, for each dataset, all vectors are inserted into the xS-tree (SR-tree), and then all vectors are used as query vectors to search through the xS-tree (SR-tree), and the averaged results are reported. For each query vector, five queries are performed to search for 1, 2, 5, 10, 20 nearest neighbors of the query vector, respectively.

All xS-trees are built with the following parameters: the distance used to calculated influence zones is 1, the storage limit for one xSquare is 240 bytes, the maximum fan-out of both non-leaf and leaf nodes is 8 and the minimum fan-out is 4, and the storage size for each non-leaf node is $2,048$ bytes. The number of levels, non-leaf nodes, and leaf nodes of the xS-trees that index the datasets with varying numbers of vectors are shown in Table 2. In all other xS-trees (indexing $1,000$

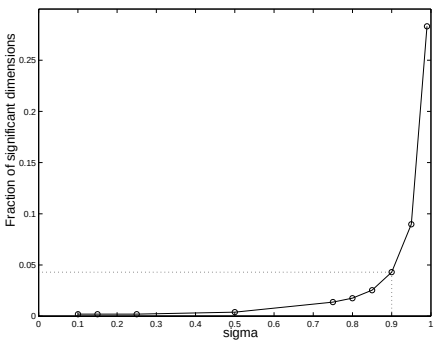
Generate a set of quasi-sparse vectors

Input	n, l, s, and f as described in the text.
Output	n quasi-sparse vectors of dimension l.
Method	(1) Generate a seed $\boldsymbol{v}$ as follows: (1.1) Let $\boldsymbol{v} = a_1 \cdots a_l$ be an l-D vector where: For $i = 1, \ldots, l$, assign a random number between 0 to 1 to a_i. (1.2) Randomly pick a set of s dimensions, denoted as D. (1.3) For each dimension k in D, do: Let a_k be $20a_k$. (1.4) Output the vector $\boldsymbol{v}$. While less than n vectors have been generated, do: (2) Exchange large values in the vector as follows: (2.1) Randomly pick a set of $f \times s$ dimensions[a], denoted D', from D. (2.2) For each dimension j in D', do (2.2.1) Randomly pick a dimension $1 \leq i \leq l$; (2.2.2) Exchange the values of a_j and a_i; (2.2.3) If $i \notin D$, remove j from D and and put i into D. (3) Modify the vector slightly as follows: For $i = 1, \ldots, l$, change value of a_i to $a_i(1 + 0.15\rho)$, where ρ is a random number between -0.5 to 0.5. (4) Output the vector $\boldsymbol{v}$. [a] If $f \times s$ is not an integer, we pick up $f \times s$ dimensions in each vectors *on average* among the total n vectors we generate. For example, if $f \times s = 0.25$, we may pick up 1 dimension once every four times when step (2) is carried out, and skip step (2) the other three times. As another example, if $f \times s = 1.5$, we may pick up 1 dimension and 2 dimensions alternatively when step (2) is executed.

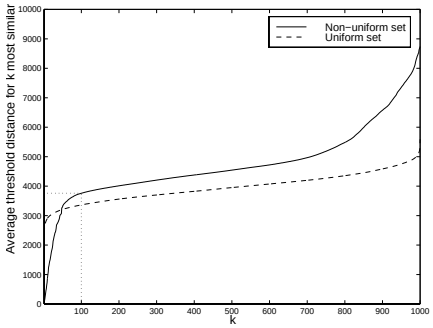
Fig. 9. Vector sets generation algorithm



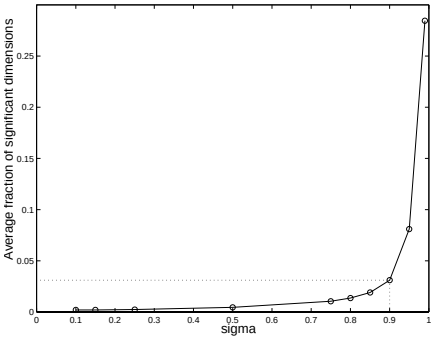
a The 501th vector



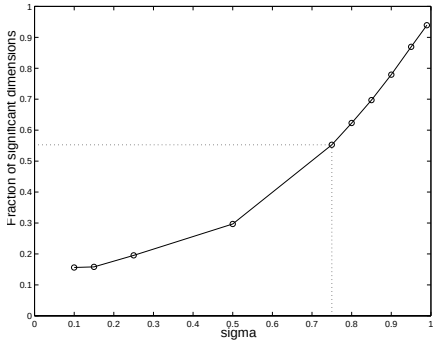
b Quasi-sparsity of (a)



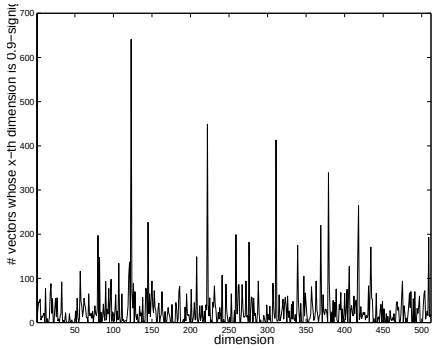
c Average threshold distances



d Average quasi-sparsity

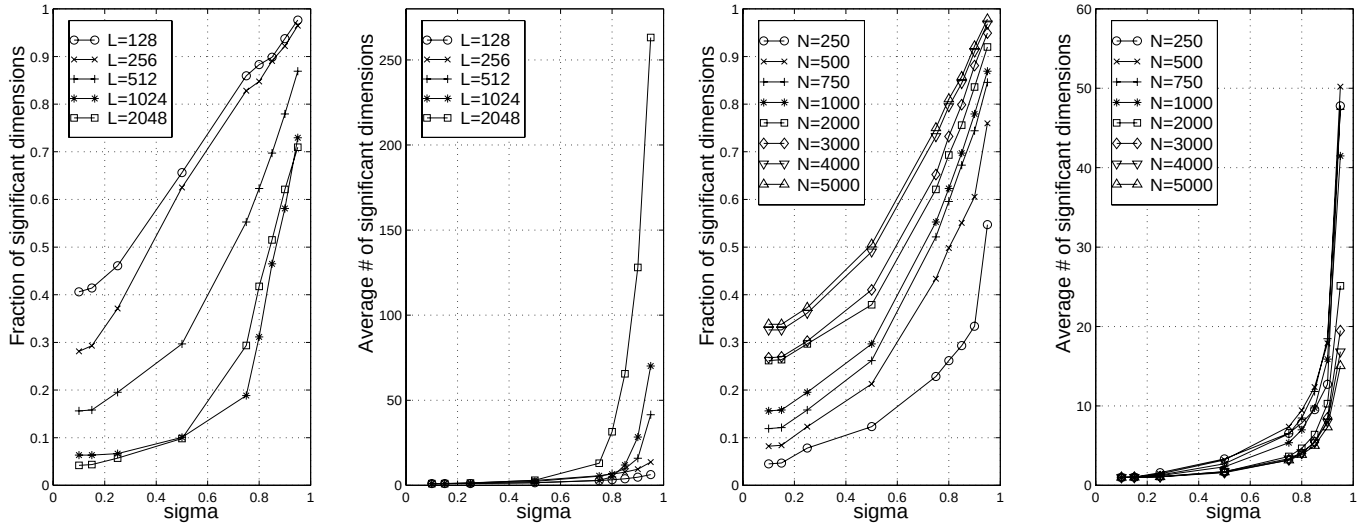


e Overall quasi-sparsity



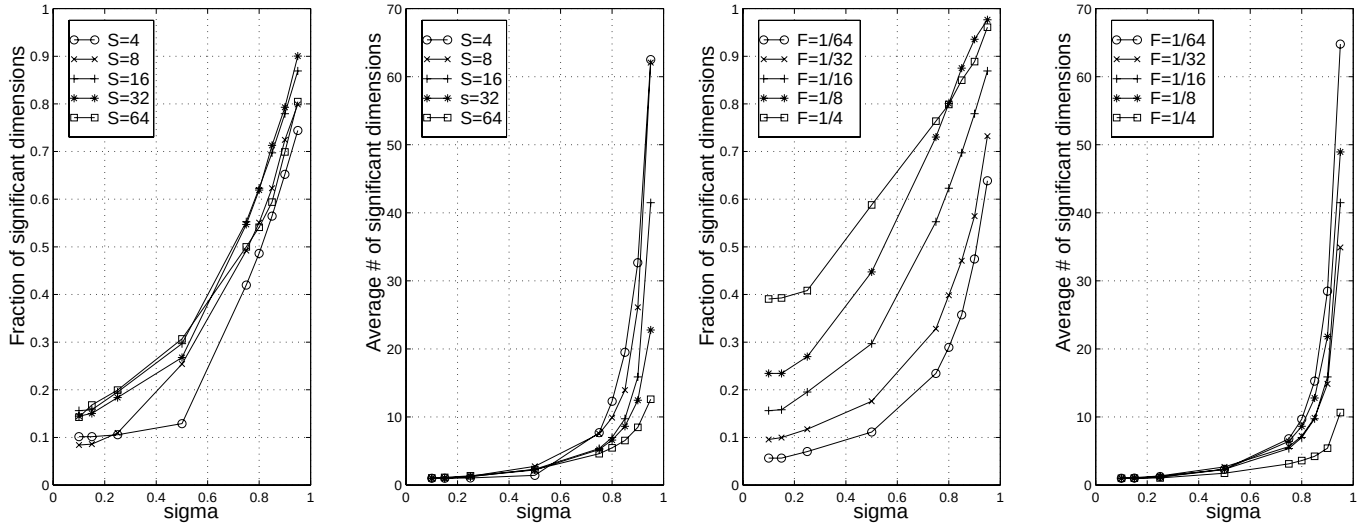
f 0.9-significant dimensions

Fig. 10a–f. A typical dataset generated with $n = 1,000$, $l = 512$, $s = 16$, and $f = \frac{1}{16}$



a Varying dimensions

b Varying number of vectors



c Varying s

d Varying c

Fig. 11a–d. Significant dimensions of datasets

Table 2. The xS-trees indexing the datasets with varying number of vectors

# of vectors	250	500	750	1,000	2,000	3,000	4,000	5,000
# of leaves	56	112	167	221	438	657	876	1093
# of non-leaves	15	30	45	60	114	174	231	288
# of levels	4	4	4	5	5	5	6	6

vectors), there are 5 levels, 57–68 non-leaf nodes, 217–247 leaf nodes.

For the SR-tree, we obtained the source code of the SR-tree from its original author[19] and extended it slightly to report some statistics. Since the SR-tree is not designed for very high-dimensional vectors, we have to use a very large node size, as shown in Table 3. These node sizes are set so that the SR-trees have the same maximum fan-out (8) for both the non-leaf nodes and the leaf nodes as the corresponding xS-trees. For example, when the dimension is 512, each branch requires $1,024$ (float) values to store the bounding rectangle and 513 (float) values to store the bounding sphere, thus, a non-leaf node requires at least $(1024+513)\times 4\times 8 = 49184$ bytes or 25 blocks. Recall that, the xS-tree instead requires only one block $(2,048)$ for each non-leaf node. For the leaf nodes, the xS-tree and the SR-tree require the same storage size.

Table 3. The node size of SR-trees

dimension	128	256	512	1,024	2,048
non-leaf node size (blocks)	7	13	25	49	97
leaf node size (blocks)	3	5	9	17	33

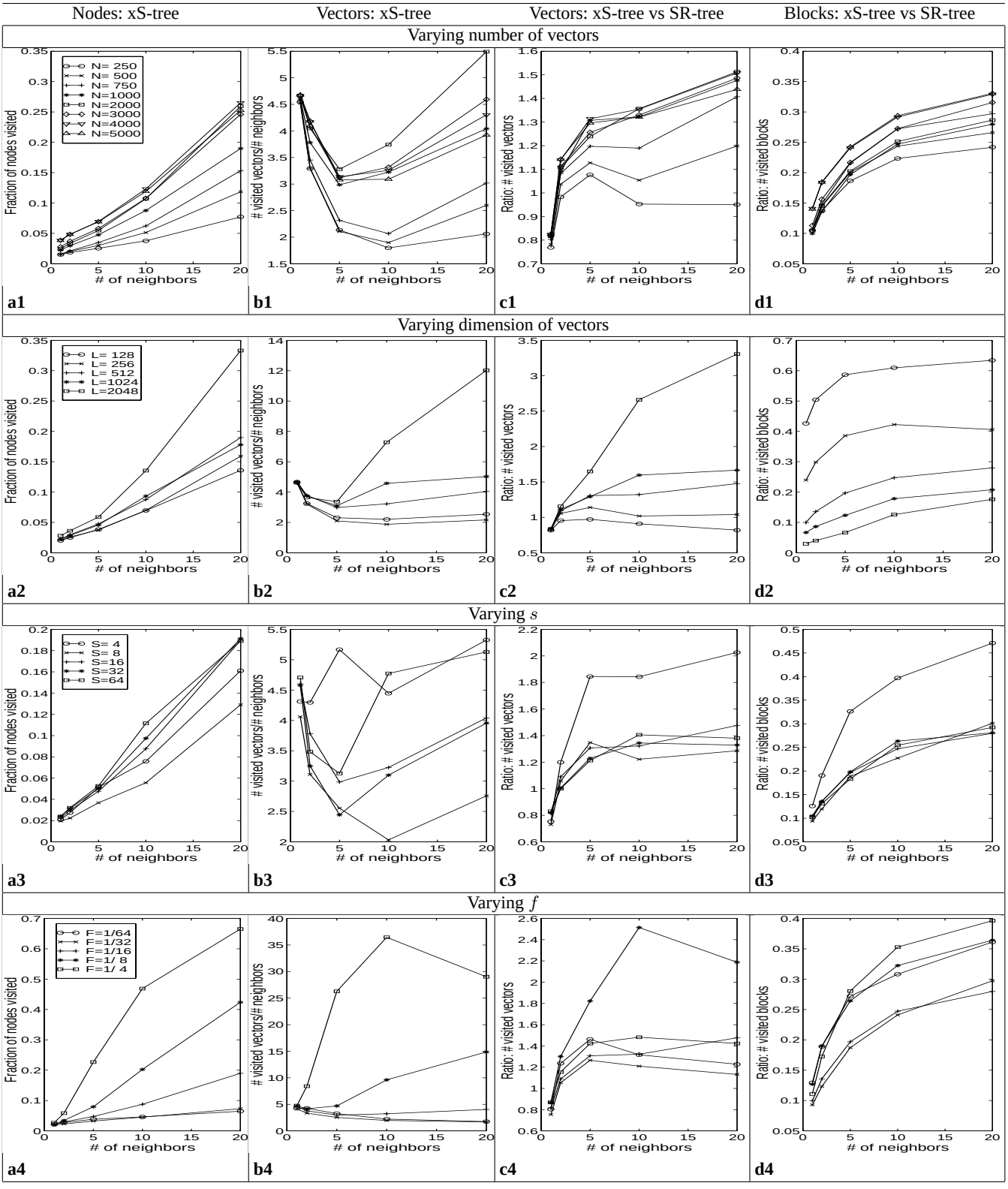


Fig. 12a1–d4. Performance of the xS-tree and comparison with the SR-tree

Figure 12 shows the averaged results of the four groups of experiments. For each experiment, we report the fraction of nodes visited during the search of the xS-tree (in Fig. 12a1–a4, the ratio of the number of vectors visited during the search of the xS-tree to the required number of nearest neighbors (i.e., $1, 2, 5, 10, 20$, respectively) (in Fig. 12b1–b4), the ratio of the number of vectors visited during the search of the xS-tree to the number of vectors visited during the search of the SR-tree (in Fig. 12c1–c4), and the ratio of the number of blocks visited during the search of the xS-tree to the number of blocks visited during the search of the SR-tree. Note that the number of blocks reported here includes the blocks storing the vectors themselves, and hence it is proportional to the total search cost when the disk accesses dominates the total cost.

Consider the first column of Fig. 12 (a1–a4); generally around 20% of nodes are visited during the search the xS-tree for 20 nearest neighbors, except for datasets with large average quasi-sparsity ($l = 2,048$ as shown in Fig. 11a) or large variations ($f = \frac{1}{4}, \frac{1}{8}$).

Consider the second column of Fig. 12 (b1–b4), the number of visited vectors during the search of the xS-tree is around 2–5 times of the required number of nearest neighbors, again, except for datasets with large average quasi-sparsity ($l = 2,048$) or large variations ($f = \frac{1}{4}, \frac{1}{8}$). Note that the absolute number of visited vectors never goes down when the number of neighbors increases, though curves (i.e., the ratios) usually goes down when the number of neighbors increase from one to five.

Clearly, the xS-tree scales well (see Fig. 12a1–b2) with both the number of vectors and the number of dimensions. Indeed, the performance of the xS-tree degenerates only when the average number (instead of the overall number) of significant dimensions or the variation of the datasets is too large.

Due to the compression of the xS-tree (and hence the enlarged bounding regions), the xS-trees usually visit 50% more vectors than the corresponding SR-trees, as shown in the third column of Fig. 12 (c1–c4). The exception in Fig. 12c2 is due to the large average quasi-sparsity ($l = 2,048$). The exception in Fig. 12c3 and c4 is due to the fact that the SR-trees on these datasets ($s = 4$, $f = \frac{1}{8}$) both have 8 child branches in their root nodes, and hence are one level lower than other SR-trees (indexing $1,000$ vectors).

However, thanks to the compression (and hence the constant node size), the xS-trees usually access only one-third of the blocks comparing to their corresponding SR-trees, as shown in the fourth column of Fig. 12 (d1–d4). In Fig. 12d2, the SR-tree performs better for datasets with lower dimensions due to the smaller node sizes. In Fig. 12d3, the exception ($s = 4$) is also due to the lower level of the SR-tree. In Fig. 12d4, the benefit of lower level of the SR-tree ($f = \frac{1}{8}$) is offset by the fact that it has the lowest utilization (58%) of leaf nodes among all SR-trees.

In comparison to the SR-tree, the xS-tree accesses less disk blocks (including the blocks storing vectors) and thus incurs less (total) search cost, though it usually visits more vectors due to the compression of the xSquares. The xS-tree outperforms the SR-tree especially when the dimension is high (Fig. 12d2).

Experiments using different distance measures, namely L_1 and L_∞, as well as using near neighbor searches as queries, are also conducted on the xS-tree. The results are comparable to what have been shown above, and are not presented here for clarity.

7 Related research

Recently, similarity search of high-dimensional feature vectors has attracted attention of many researchers due to its practical importance. A significant contribution to this area is paper [1] where the discrete Fourier transform is used to extract the features from time series. For time series like stock prices, only a few lowest-frequency features are most dominant, and the projections of vectors into these dimensions can be used as a filter. The indexing structure is built on the projected vectors to prune the searching space. Other authors, e.g., [13,23], extend this idea to index time series under different similarity measures. In general, these works exploit the quasi-sparsity of the Fourier transform for many time series. However, they are limited to applications where the quasi-sparse feature vectors (derived from the Fourier transform) all have their significant values located in a specific set of dimensions. In contrast, the xS-tree handles quasi-sparse vector sets in which significant values may appear at different dimensions for different vectors.

There are also a number of data structures proposed to directly index high-dimensional points for similarity searches, see [6] for a recent survey. Examples include the SS-tree [29], the SR-tree [18], the TV-tree [21], and the Pyramid tree. Without the quasi-sparsity assumption and the compression technique, the dimension of points that the SS-tree and the SR-tree can handle is usually in tens (otherwise the node size can be extremely large, as discussed in Sect. 6). In contrast, the xS-tree is developed to handle quasi-sparse vectors with hundreds of dimensions.

The TV-tree [21] uses an interesting idea where at different levels, different dimensions are used in the bounding regions, and can potentially handle very high-dimensional vectors. However, to choose appropriate dimensions for different levels, the TV-tree assumes that the relative importance of the dimensions is known a priori. In contrast, designed as a general indexing structure for quasi-sparse vectors, the xS-tree dynamically decides significant dimensions when points are inserted or deleted via the compression process.

The pyramid technique, proposed in [3], is based on a special data space partitioning strategy tuned for high-dimensional range queries. Though scaled well with the dimension, it cannot handle strong skew (which is the case for quasi-sparse vectors handled by the xS-tree) very well.

The paper [15] proposes applying the LSD-tree directly to some very high-dimensional vectors, more specifically, normalized feature vectors of documents. However, the tree could be arbitrarily unbalanced and may degenerate to a chain. To avoid this problem, [15] exploits the distribution of the vectors and design a specific clustering principle. In contrast, adapted from the R-tree, the xS-tree is a balanced structure.

The concept of "Minkowski Sum", which is similar to the concept of "influence zone" proposed in this paper, has been discussed in the study of cost model of nearest neighbor searches [5]. However, we use the concept of influence zone in studying clustering strategies. In addition, we present a more

general result for calculating the influence zone, allowing the family of L_p distances.

Distance-based trees [7,27,8,9] are also used to support similarity searches. Proposed for a more general metric-space[4], these trees cluster objects only based on their relative distances. Without any information of the geometry of the data points, the bounding regions are spheres centered at some data points. When the data points are high-dimensional quasi-sparse vectors, the spheres that bound vectors with different significant dimensions would be extremely large. Furthermore, the center point of the sphere, which is a high-dimensional vector, may also require large storage space.

Other related works include researches on feature extraction of complex objects. A variety of mathematical transforms, such as discrete cosine transform and single value decomposition [10], have been proposed to extract feature vectors. Among them, the wavelet transforms attract more and more attention, especially for similarity searches of images [16, 28]. The feature vectors extracted through these transforms are often high-dimensional and quasi-sparse. Traditional spatial access methods can be used as filters when these vectors have similar energy distributions. However, when these vectors have dissimilar energy distributions, our work can be used.

8 Conclusion

In this paper, we studied the feasibility of adapting spatial access methods to index high-dimensional quasi-sparse vectors with dissimilar energy distributions. In particular, we introduced the xS-tree, adapted from the R-tree, to help the fast identification of similar quasi-sparse vectors. The quasi-sparsity was exploited to overcome the limitation of the R-tree in indexing very high-dimensional vectors. An improved clustering principle was also developed.

Experiments showed that the xS-tree performs and scales well for high-dimensional yet sparse or quasi-sparse vector sets when these vectors are geometrically clustered. Actually, when the vectors are well clustered and sparse, the number of significant dimensions of the bounding xSquares is expected to be quite small for the nodes at several bottom levels, hence the error introduced by the lossy compression is expected to be small. The reason is that the points clustered together have the property that most of their significant dimensions are in common. For nodes at higher levels, the number of significant dimensions of their bounding xSquare can be very large, and hence compression may give more error. However, since the total number of high level nodes is very small comparing to the total number of nodes in the tree, the performance does not severely degenerate. In other words, the xS-tree does not suffer the "dimensionality curse" when indexing the vectors which it is designed for.

In this paper, instead of real datasets, we used synthetic datasets in our experiments. By generating datasets ourselves, we can control the characteristics of the datasets, examine the performance of the xS-tree by using datasets with varying characteristics, and hence study the xS-tree in a comprehensive fashion. We also compare the xS-tree with the SR-tree, one of the latest indexing structures designed for high-dimensional vectors. However, the SR-tree is not prepared for several hundred dimensions, and requires very large non-leaf nodes. Though the xS-tree visits more vectors in the leaf nodes due to the lossy compression of the xSquares, it requires significantly fewer numbers of total disk block accesses thanks to the compression of the xSquares.

There are several future directions worth pursuing. First, as mentioned in Sect. 4, the volume of influence zone can be extended to a distribution function. Furthermore, for similarity searches of points, probabilistic models for the performance of indexing structures can be built using the concept of influence zone. It will be interesting to study such probabilistic models and their effect on the xS-tree. Second, while the R-tree is adopted as the starting point of the xS-tree, other SAMs, especially the hB-tree, can also be adapted for indexing high-dimensional vectors. Related interesting future direction will be modifying distance-based trees for this purpose. The distance-based trees save storage space by using data points as the center points of bounding regions and this can be extended to using some combination of data points. Finally, in addition, distance measures other than L_p norms, such as Hamming distance for bit vectors, can also be used. Whether the xS-tree works with these distance measures, or more generally, the characteristics of the distance measures that the xS-tree can work with, is also an open problem.

Acknowledgements. The authors wish to thank the anonymous reviewers as well as the editors for their helpful comments.

A Appendix

A.1 Proof of Theorem 1

Proof.

Step 1: The influence zone of R, by definition, is

$$IZ(R,d) = \left\{ \mathbf{y} = y_1 \cdots y_l | \exists \mathbf{x} = x_1 \cdots x_l \right.$$
$$\left. \left(\forall 1 \le i \le l (0 \le x_i \le a_i) \wedge \left(\sum_{i=1}^{l} (|x_i - y_i|)^p \right)^{\frac{1}{p}} \le d \right) \right\}$$

It can be partitioned into $l+1$ parts $P_0, \ldots, P_l$ where:

$$P_0 = \{ \mathbf{y} = y_1 \cdots y_l | \forall 1 \le i \le l (0 \le y_i \le a_i) \},$$
$$P_i = \left\{ \mathbf{y} = y_1 \cdots y_l | \exists 1 \le k_1 < \cdots < k_i \le l \left(\forall 1 \le j \le l \right. \right.$$
$$(j \notin \{k_1, \ldots, k_i\} \rightarrow 0 \le y_j \le a_j) \wedge$$
$$\forall 1 \le j \le i (y_{k_j} < 0 \vee y_{k_j} > a_{k_j}) \wedge$$
$$\exists \mathbf{x} = x_1 \cdots x_l \left(\forall 1 \le i \le l (0 \le x_i \le a_i) \wedge \right.$$
$$\left. \left. \left. \left(\sum_{i=1}^{l} (|x_i - y_i|)^p \right)^{\frac{1}{p}} \le d \right) \right) \right\}$$

[4] In a metric space, the distance function is required to be positive, symmetric, and satisfy the triangle inequality, see [24] for details.

for $i=1,\ldots,l$.

Intuitively, the influence zone is partitioned by restricting the range of coordinates in a number of dimensions. It is clear that $P_i \subset IZ(R,b)$ for $i=0,1,\ldots,l$. Thus, $\bigcup_{i=0}^{l} P_i \subset IZ(R,d)$. In addition, for any $\mathbf{y} \in IZ(R,d)$, by definition, $\mathbf{y} \in P_i$ if $\mathbf{y}$ has i dimensions such that its value in each of these dimensions is out of the projection of the rectangle R at that dimension. In addition, $P_j \cap P_k = \emptyset$ for $0 \le j \ne k \le l$. Hence, the partition is sound and complete, i.e., no two groups in the partition overlaps and any $\mathbf{y}$ in the influence zone must fall into one group of the partition.

Step 2: For $i=1,\ldots,l$, let

$$P_i' = \left\{ \mathbf{y} = y_1 \cdots y_l \middle| \exists 1 \le k_1 < \cdots < k_i \le l \left(\forall 1 \le j \le l \right.\right.$$
$$(j \notin \{k_1,\ldots,k_i\} \rightarrow 0 \le y_j \le a_j) \wedge$$
$$\exists z_{k_1},\ldots,z_{k_i} \left(\forall 1 \le j \le i \left((y_{k_j} < 0 \wedge z_{k_j} = 0) \right.\right.\right.$$
$$\left.\left.\left. \vee (y_{k_j} > a_{k_j} \wedge z_{k_j} = a_{k_j}) \right) \wedge \left(\sum_{j=1}^{i} (|z_{k_j} - y_{k_j}|)^p \right)^{\frac{1}{p}} \le d \right) \right) \right\}.$$

Then $P_i = P_i'$ for $i=1,\ldots,l$. Indeed, we just removed some constraints that are implied by others. More specifically, for those out-of-range dimensions, x is chosen as either of the two ends of the range. This can be formally proved as follows:

For each $\mathbf{y} = y_1 \cdots y_l \in P_i$, by definition, we have $1 \le k_1 < \cdots < k_i \le l$ such that: 1) $0 \le y_j \le a_j$ for $1 \le j \le l$ and $j \notin \{k_1,\ldots,k_i\}$; 2) $y_{k_j} < 0 \vee y_{k_j} > a_{k_j}$ for $j=1,\ldots,i$; and 3) there exists $\mathbf{x} = x_1 \cdots x_l$ satisfying $\forall 1 \le i \le l (0 \le x_i \le a_i)$ and $\left(\sum_{i=1}^{l} (|x_i - y_i|)^p\right)^{\frac{1}{p}} \le d$.

Let $1 \le j \le i$. If $y_{k_j} < 0$, let $z_{k_j} = 0$. Then $(|y_{k_j} - z_{k_j}|)^p < (|y_{k_j} - x_{k_j}|)^p$ (because $x_{k_j} \ge 0$). If $y_{k_j} > a_{k_j}$, let $z_{k_j} = a_{k_j}$. Then $(|y_{k_j} - z_{k_j}|)^p < (|y_{k_j} - x_{k_j}|)^p$ (because $x_{k_j} \le a_{k_j}$). Thus,

$$\left(\sum_{j=1}^{i} (|y_{k_j} - z_{k_j}|)^p \right)^{\frac{1}{p}} \le \left(\sum_{j=1}^{i} (|y_{k_j} - x_{k_j}|)^p \right)^{\frac{1}{p}}$$
$$\le \left(\sum_{j=1}^{l} (|y_j - x_j|)^p \right)^{\frac{1}{p}}$$
$$\le d.$$

Hence $\mathbf{y} \in P_i'$.

On the other hand, for each $\mathbf{y} = y_1 \cdots y_l \in P_i'$, it is easy to show that $\mathbf{y} = y_1 \cdots y_l \in P_i$ by constructing $\mathbf{z}$ as follows:

$$z_i = \begin{cases} 0 & \text{if } y_j < 0, \\ a_j & \text{if } y_j > a_j, \\ y_j & \text{otherwise.} \end{cases}$$

Step 3: For $1 \le k_1 < \cdots < k_i \le l$, the volume of

$$S_i(k_1,\ldots,k_i) = \{\bar{\mathbf{y}} = y_{k_1} \cdots y_{k_i} | \exists z_{k_1},\ldots,z_{k_i} \left(\forall 1 \le j \le i \right.$$
$$((y_{k_j} < 0 \wedge z_{k_j} = 0)$$
$$\vee (y_{k_j} > a_{k_j} \wedge z_{k_j} = a_{k_j}))$$
$$\left. \wedge \left(\sum_{j=1}^{i} (|z_{k_j} - y_{k_j}|)^p \right)^{\frac{1}{p}} \le d \right) \Bigg\}$$

is the volume of i-D query region, i.e., $\mathbf{Q}_i = \int_{q_i} \mathrm{d}\bar{\mathbf{y}}$ where $q_i = \left\{ \left(\sum_{j=1}^{i} (|y_{k_j}|)^p \right)^{\frac{1}{p}} \le d \right\}$. We show this as follows:

for $1 \le j \le i$, let f_j, g_j, and h_j be the conditions:

$$\begin{array}{ll} f_j: & y_{k_j} < 0 \wedge z_{k_j} = 0, \\ g_j: & y_{k_j} > a_{k_j} \wedge z_{k_j} = a_{k_j}, \\ h_j: & f_j \vee g_j. \end{array}$$

In addition, let h_0 be the condition $\left(\sum_{j=1}^{i} (|z_{k_j} - y_{k_j}|)^p \right)^{\frac{1}{p}} \le d$. The volume of $S_i(k_1,\ldots,k_i)$ is $\int_{\exists z_{k_1},\ldots,z_{k_i} \left(\bigwedge_{j=0}^{i} h_j \right)} \mathrm{d}\bar{\mathbf{y}}$.

For each j, f_j, and g_j cannot be true at the same time, i.e., the part of $S_i(k_1,\ldots,k_i)$ constrained by f_i will not intersect with that constrained by g_j. Thus, the above volume can be the sum of 2^i parts, i.e.,

$$\sum_{\theta_j \in \{f_j, g_j\}} \int_{\exists z_{k_1},\ldots,z_{k_j} \left(h_0 \wedge \bigwedge_{j=1}^{i} (\theta_j) \right)} \mathrm{d}\bar{\mathbf{y}}.$$

However, the volume of each part is independent of a_1, …, a_l. This is because for each $1 \le m \le i$,

$$\begin{array}{rl} & \displaystyle\int_{\exists z_{k_1},\ldots,z_{k_j} \left(h_0 \wedge g_m \wedge \bigwedge_{j \ne m} (\theta_m) \right)} \mathrm{d}\bar{\mathbf{y}} \\ = & \displaystyle\int_{\exists z_{k_1},\ldots,z_{k_j} \left(\bigwedge_{j \ne m} (\theta_m) \wedge \left(\sum_{j=1}^{i} (|z_{k_j} - y_{k_j}|)^p \right)^{\frac{1}{p}} \right.} \\ \le & \left. d \wedge y_{k_m} > a_{k_m} \wedge z_{k_m} = a_{k_m} \right) \mathrm{d}\bar{\mathbf{y}} \\ \overset{y_{k_m} - a_{k_m} \to y_{k_m}}{=} & \displaystyle\int_{\exists z_{k_1},\ldots,z_{k_j} \left(\bigwedge_{j \ne m} (\theta_m) \wedge \left(\sum_{j=1}^{i} (|z_{k_j} - y_{k_j}|)^p \right)^{\frac{1}{p}} \right.} \\ \le & \left. d \wedge y_{k_m} > 0 \wedge z_{k_m} = 0 \right) \mathrm{d}\bar{\mathbf{y}}. \end{array}$$

By using the above step for $1 \le j \le i$, the volume of $S_i(k_1,\ldots,k_i)$ becomes:

$$\int_{\exists z_{k_1},\ldots,z_{k_j} \left(h_0 \wedge \bigwedge_{j=1}^{i} y_{k_j} \ne 0 \wedge \bigwedge_{j=1}^{i} z_{k_j} = 0 \right)} \mathrm{d}\bar{\mathbf{y}}$$
$$= \int_{q_i \wedge \exists z_{k_1},\ldots,z_{k_j} \left(\bigwedge_{j=1}^{i} y_{k_j} \ne 0 \right)} \mathrm{d}\bar{\mathbf{y}}.$$

The only difference between this and $\mathbf{Q}_i$ is

$$\int_{q_i \wedge \exists z_{k_1},\ldots,z_{k_j} \left(\bigvee_{j=1}^{i} y_{k_j} = 0 \right)} \mathrm{d}\bar{\mathbf{y}},$$

which is known to be 0.

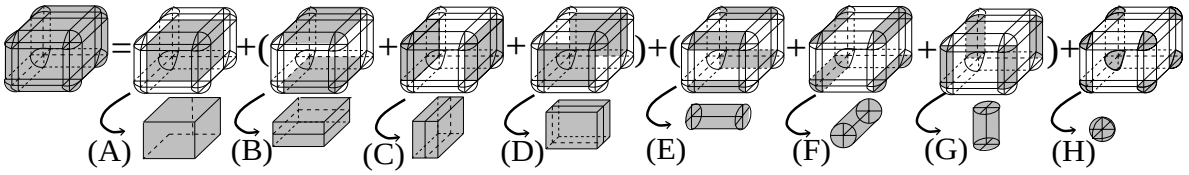


Fig. 13. The volume of the influence zone of a 3-D rectangle

Step 4: For $1 \le k_1 < \cdots < k_i \le l$, the volume of $T_i(k_1, \ldots, k_l)$ is

$$\left\{ \mathbf{y} = y_1 \cdots y_l | \forall 1 \le j \le l (j \notin \{k_1, \ldots, k_i\} \right.$$
$$\left. \rightarrow 0 \le y_j \le a_j) \wedge \right.$$
$$\exists \mathbf{z} = z_1 \cdots z_l \left(\forall 1 \le j \le i \left((y_{k_j} < 0 \wedge z_{k_j} = 0) \right.\right.$$
$$\left.\left. \vee (y_{k_j} > a_{k_j} \wedge z_{k_j} = a_{k_j}) \right) \wedge \right.$$
$$\left. \left(\sum_{j=1}^{i} (|z_{k_j} - y_{k_j}|)^p \right)^{\frac{1}{p}} \le d \right) \right\}$$

is $\mathbf{Q}_i \prod_{j \notin \{k_1, \ldots, k_i\}} a_j$. To show this, we divide the l dimensions into $(l - i + 1)$ sets: $\{k_1, \ldots, k_i\}$, and $\{j\}$ for $j \notin \{k_1, \ldots, k_i\}$. Clearly, there is no constraint that connects any two sets of dimensions. Thus, the total volume is the product of the $(l - i + 1)$ parts.

Step 5: Thus the total volume of $P_i{}'$ is the sum of $T_i(k_1, \ldots, k_l)$ for all possible combination of $k_1, \ldots, k_l$, i.e.,

$$\sum_{1 \le k_1 < \cdots < k_i \le l} \left(\mathbf{Q}_i \prod_{j \notin \{k_1, \ldots, k_i\}} a_j \right)$$
$$= \mathbf{Q}_i \sum_{1 \le k_1 < \cdots < k_i \le l} \prod_{j \notin \{k_1, \ldots, k_i\}} a_j$$
$$= \mathbf{Q}_i \mathbf{R}_i.$$

The total volume of $IZ(R, d)$ is the sum of the volume of P_0 and $P_i{}'$ for $i = 1, \ldots, l$. Note that P_0 is exactly the rectangle R itself, and the volume of R is given by $\mathbf{R}_0$. To finish the proof, notice that $\mathbf{Q}_0$ is defined as 1.

Figure 13 illustrates the above process in the 3-dimensional case. The 3-D influence zone is broken into four parts: $P_0 = (A)$, $P_1 = (B) + (C) + (D)$, $P_2 = (E) + (F) + (G)$, and $P_4 = (H)$. The distance measure used here is the L_2 norm. The query regions are an interval, a circle, and a ball, respectively, in the 1-, 2- and 3-D spaces.

A.2 Proof of Theorem 2

Proof. From Theorem 1, the volume of $IZ(R_1, d)$ is: $\sum_{i=0}^{l} \mathbf{R}_i \mathbf{Q}_i$, where $\mathbf{R}_i = \sum_{\forall 1 \le k_1 < \cdots < k_i \le l} \prod_{j \notin \{k_1, \ldots, k_i\}} a_j$, similarly the volume of $IZ(R_2, d)$ is: $\sum_{i=0}^{l} \mathbf{R}_i{}' \mathbf{Q}_i$, where $\mathbf{R}_i{}' = \sum_{\forall 1 \le k_1 < \cdots < k_i \le l} \prod_{j \notin \{k_1, \ldots, k_i\}} a_j{}'$.

For each $1 \le i \le l$, $\mathbf{R}_i$ can be rewritten as $a_{m_1} a_{m_2} A + (a_{m_1} + a_{m_2}) B + C$ where A, B and C are expressions without a_{m_1} or a_{m_2} (a_{m_1} and a_{m_2} have the same coefficients B because the expression is symmetric in terms of a_{m_1} and a_{m_2}). Similarly, $\mathbf{R}_i{}'$ can be rewritten as $a_{m_1}{}' a_{m_2}{}' A' + (a_{m_1}{}' + a_{m_2}{}') B' + C'$. Since $a_i{}' = a_i$ for $1 \le i \le l$ and $i \ne m_1$ and $i \ne m_2$, $A = A'$, $B = B'$ and $C = C'$.

In addition,

$$\begin{aligned} &|a_{m_1} - a_{m_2}| > |a'_{m_1} - a'_{m_2}| \qquad \text{(given)} \\ &\Rightarrow (a_{m_1} - a_{m_2})^2 > (a'_{m_1} + a'_{m_2})^2 \\ &\Rightarrow (a_{m_1} + a_{m_2})^2 > (a'_{m_1} + a'_{m_2})^2 \text{(as } a_{m_1} a_{m_2} = a_{m_1}{}' a_{m_2}{}') \\ &\Rightarrow a_{m_1} + a_{m_2} \ge a_{m_1}{}' + a_{m_2}{}' \quad \text{(as values are positive)} \end{aligned}$$

Therefore, $\mathbf{R}_i \ge \mathbf{R}_i{}'$ for any $1 \le i \le l$. Furthermore, notice $\mathbf{R}_0 = \mathbf{R}_0{}'$. Thus, the volume of $IZ(R_1, d)$ is greater than that of $IZ(R_2, d)$.

A.3 Best merge-only strategy

To compress an xSquare by applying a series of merging-only steps, the best strategy that results in the least enlargement of the volume of the xSquare's influence zone will only contain steps that merging consecutive squares, when squares are sorted by their radii. To show this, we prove the following proposition:

Proposition 1. *Given a l-D xSquare consisting of three squares: S_1 in D_1 with side length $2r_1$, S_2 in D_2 with side length $2r_2$, and S_3 in D_3 with side length $2r_3$ (thus, $D_1 \cup D_2 \cup D_3 = \{1, \ldots, l\}$, and $D_1 \cap D_2 = D_2 \cap D_3 = D_3 \cap D_1 = \emptyset$), such that $r_1 \le r_2 \le r_3$, the enlargement of the volume of the xSquare's influence zone introduced by merging S_1 and S_3 is greater than that introduced by merging S_1 and S_2.*

Proof. Let $d_1 = |D_1|$, $d_2 = |D_2|$, and $d_3 = |D_3|$. Let $X_{i,j}$ be the result xSquare by merging S_i and S_j where $i < j$, and let $V_{i,j}$ be the volume of $X_{i,j}$. Thus, $V_{1,2} = r_2^{d_1 + d_2} r_3^{d_3}$ and $V_{1,3} = r_3^{d_1 + d_3} r_2^{d_2}$.

For any $D \subset \{1, \ldots, l\}$, let $X_{i,j}^D$ be the projection of $X_{i,j}$ into dimension set D, and $V_{i,j}^D$ be the volume of $X_{i,j}^D$. Let $d_1^D = |D_1 \cap D|$, $d_2^D = |D_2 \cap D|$, and $d_2^D = |D_2 \cap D|$. Then, $V_{1,2}^D = r_2^{d_1^D + d_2^D} r_3^{d_3^D}$ and $V_{1,3}^D = r_3^{d_1^D + d_3^D} r_2^{d_2^D}$.

$$\frac{V_{1,3}^D}{V_{1,2}^D} = \left(\frac{r_3}{r_2} \right)^{d_1^D} \ge 1.$$

Since the volume of the influence zone of an xSquare is decided by the query region and all possible projections of the xSquare (see Theorem 1), the influence zone of $X_{1,3}$ would be greater than that of $X_{1,2}$.

It is straightforward to extend the above proposition to xSquares with more than three squares.

References

1. Agrawal R, Faloutsos C, Swami AN (1993) Efficient similarity search in sequence databases. In: The 4th Int. Conf. on Foundations of Data Organization and Algorithms, pp 69–84, Evanston, Ill., October 1993
2. Beckmann N, Kriegel H-P, Schneider R, Seeger B (1990) The R*-tree: an efficient and robust access method for points and rectangles. In: Proc. of ACM SIGMOD Int. Conf. on Management of Data, May 1990
3. Berchtold S, Böhm C, Kriegel H-P (1998) The pyramid-technique: towards breaking the curse of dimensionality. In: Proc. 1998 ACM SIGMOD, pp 142–153, Seattle, Washington, June 1998
4. Berchtold S, Keim DA, Kriegel H-P (1996) The X-tree: an index structure for high-dimensional data. In: Proc. 22th Int. Conf. on Very Large Data Bases, pp 28–39, Mumbai (Bombay), India, 1996
5. Berchtold S, Böhm C, Keim DA, Kriegel H-P (1997) A cost model for nearest neighbor search in high-dimensional data space. In: PODS Conference, pp 78–86
6. Berchtold S, Keim DA (1998) High-dimensional index structures, database support for next decade's applications (tutorial). In: SIGMOD Conference, 1998
7. Bozkaya T, Özsoyoglu ZM (1997) Distance-based indexing for high-dimensional metric spaces. In: Proc. ACM SIGMOD, pp 357–368
8. Brin S (1995) Nearest neighbor search in large metric space. In: Proc. 21th Int. Conf. on Very Large Data Bases, pp 574–584, Zurich, Switzerland
9. Ciaccia P, Patella M, Zezula P (1997) M-tree: an efficient access method for similarity search in metric spaces. In: Proc. 23th Int. Conf. on Very Large Data Bases, pp 426–435, Athens, Greece
10. Faloutsos C, Lin K (1995) Fastmap: a fast algorithm for indexing, data mining and visualization of traditional and multimedia datasets. In: Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data, pp 163–174, San Jose, Calif.
11. Faloutsos C, Ranganathan M, Manolopoulos Y (1994) Fast subsequence matching in time-series databases. In: Proc. ACM SIGMOD, pp 419–429
12. Gaede V, Günther O (1995) Survey on multidimensional access methods. Technical Report ISS-16, William Penney Laboratory, Imperial College, London, UK, August 1995
13. Goldin DQ, Kanellakis PC (1995) On similarity queries for time-series data: constraint specification and implementation. In: Proc. Int. Conf. on Principles and Practice of Constraint Programming, pp 137–153
14. Guttman A (1984) R-trees: a dynamic index structure for spatial searching. In: Proc. ACM SIGMOD, pp 47–57, June 1984
15. Henrich A (1996) Adapting a spatial access structure for document representations in vector space. In: Proc. Conference on Information and Knowledge Management, pp 19–26, Rockville, Md.
16. Jacobs CE, Finkelstein A, Salesin DH (1995) Fast multiresolution image querying. In: Proc. SIGGRAPH, 1995. Also appeared as Technical Report 95-01-06, Department of Computer Science and Engineering, University of Washington
17. Kamel I, Faloutsos C (1994) Hilbert R-tree: an improved R-tree using fractals. In: Proc. 20th Int. Conf. on Very Large Data Bases
18. Katayama N, Satoh S (1997) The SR-tree: an index structure for high-dimensional nearest neighbor queries. In: Proc. ACM SIGMOD, pp 369–380
19. Katayama N (2000) Norio Katayama's Home Page (English). http://www.rd.nacsis.ac.jp/∼katayama/homepage
20. Kowalski G (1997) Information retrieval systems, theory and implementation. Kluwer Academic, Boston, Mass.
21. Lin KI, Jagadish HV, Faloutsos C (1994) The TV-tree: An index structure for high-dimensional data. VLDB J 3(4): 517–542
22. Lomet DB, Salzberg B (1990) The hB-tree: a multiattribute indexing method with good guaranteed performance. ACM Trans Database Syst 15: 625–658
23. Rafiei D, Mendelzon AO (1997) Similarity-based queries for time series data. In: Proc. ACM SIGMOD, pp 13–25, May 1997
24. Roman S (1992) Advanced linear algebra. Springer, Berlin Heidelberg New York
25. Roussopoulos N, Kelley S, Vincent F (1995) Nearest neighbor queries. In: Proc. of ACM-SIGMOD Intl. Conf. on Management of Data
26. Sellis TK, Roussopoulos N, Faloutsos C (1987) The R+-Tree: a dynamic index for multi-dimensional objects. In: 13th Int. Conf. on Very Large Data Bases, pp 507–518, Brighton, UK
27. Uhlmann JK (1991) Satisfying general proximity/similarity queries with metric trees. Inf Process Lett 40(4): 175–179
28. Wang JZ, Wiederhold G, Firschein O, Wei SX (1997) Wavelet-based image indexing techniques with partial sketch retrieval capability. In: Proc. 4th Forum on Research and Technology Advances in Digital Libraries
29. White DA, Jain R (1996) Similarity indexing with the SS-tree. In: Proc. Int. Conf. on Data Engineering, pp 516–523