VLDB Journal (2002) 11: 93–108 / Digital Object Identifier (DOI) 10.1007/s00778-002-0066-9

Spatial indexing of high-dimensional data based on relative approximation

Yasushi Sakurai[1], **Masatoshi Yoshikawa**[2], **Shunsuke Uemura**[2], **Haruhiko Kojima**[1]

[1] NTT Cyber Space Laboratories, 1-1 Hikari-no-oka, Yokosuka, Kanagawa 239-0847, Japan; e-mail: sakurai.yasushi@lab.ntt.co.jp, kojima@aether.hil.ntt.co.jp

[2] Nara Institute of Science and Technology, 8916-5 Takayama, Ikoma, Nara 630-0101, Japan; e-mail: {yosikawa, uemura}@is.aist-nara.ac.jp

Edited by T. Sellis. Received: December 8, 2000 / Accepted: March 20, 2002
Published online: September 25, 2002 – © Springer-Verlag 2002

Abstract. We propose a novel index structure, the A-tree (approximation tree), for similarity searches in high-dimensional data. The basic idea of the A-tree is the introduction of virtual bounding rectangles (VBRs) which contain and approximate MBRs or data objects. VBRs can be represented quite compactly and thus affect the tree configuration both quantitatively and qualitatively. First, since tree nodes can contain a large number of VBR entries, fanout becomes large, which increases search speed. More importantly, we have a free hand in arranging MBRs and VBRs in the tree nodes. Each A-tree node contains an MBR and its children VBRs. Therefore, by fetching an A-tree node, we can obtain information on the exact position of a parent MBR and the approximate position of its children. We have performed experiments using both synthetic and real data sets. For the real data sets, the A-tree outperforms the SR-tree and the VA-file in all dimensionalities up to 64 dimensions, which is the highest dimension in our experiments. Additionally, we propose a cost model for the A-tree. We verify the validity of the cost model for synthetic and real data sets.

Keywords: High-dimensional data – Similarity search – Relative approximation

1 Introduction

1.1 Data retrieval in high-dimensional space

Fast content-based retrieval is a core function in providing a high-quality human interface for large-scale multimedia databases. Most content-based retrieval schemes use feature vectors extracted from multimedia data as keys. For instance, the features extracted from images include color, texture, and structure. In the balance of the reduction of computational cost and the increase of object recognition ratio, feature vectors of dimensionality ten or higher are used in many recognition methods [16] [17] and retrieval systems [23]. Since retrieving high-dimensional feature vectors incurs high cost for large data sets, new spatial indices and search methods that offer efficient data retrieval are required. Various spatial indices have been proposed [20] [8]. This paper introduces a new index structure, called A-tree (approximation tree) [21], that offers significantly higher search performance than existing indices.

A short and earlier version of this paper has appeared in the proceedings of the 26th International Conference on Very Large Data Bases (VLDB 2000) [21].

1.2 Related work

The conventional approach to supporting similarity searches in high-dimensional vector spaces can be broadly classified into two categories. The first approach uses data-partitioning index trees. Neighboring vectors are covered by MBRs (minimum bounding rectangles) or MBSs (minimum bounding spheres), which are organized in a hierarchical tree structure. Many index tree schemes have been proposed. They include the R-tree [10], the R*-tree [6], the Hilbert R-tree [13], and the SS-tree [22]. In addition, nearest neighbor search methods using such indices have been proposed [11] [12] [18]. Two recently proposed indices, the X-tree [5] and the SR-tree [15], are reported to offer good performance. The X-tree [5] introduces the notion of a supernode and outperforms the R*-tree. The SR-tree [15] has a unique feature in that it uses both MBRs and MBSs and is reported to outperform both the R*-tree and the SS-tree. The second approach is the use of approximation files. Among the others, the VA-file (vector approximation file) [24] is a simple yet powerful scheme. The VA-file divides the data space into cells and allocates a bit-string to each cell. The vectors inside a cell are approximated by the cell, and the VA-file itself is simply an array of these geometric approximations. When searching vectors, the entire VA-file is scanned to select candidate vectors. Those candidates are then verified by visiting the vector files. In [24], Weber et al. report that the VA-file outperforms both the R*-tree and the X-tree when the dimensionality is high.

In the field of spatial search of high-dimensional data, the notorious “curse of dimensionality" problem looms large. Of late, search methods that present an approximate answer [1] [9], have been proposed to avoid the problem. Although these methods are useful, the goal of our work is to overcome the problem by developing a search method that gives an exact answer.

1.3 Our contributions

The introduction of the A-tree is motivated by a comparison and analysis of the SR-tree and the VA-file. Since no result on the comparison between these two access methods is available, we first performed experiments comparing the two access methods. The results of the experiments were used to develop the A-tree structure and its search and update algorithms. The basic idea of A-tree is the introduction of virtual bounding rectangles (VBRs), which contain and approximate MBRs or data objects. Although other spatial indices that combine a tree structure and a quantization technique have been proposed [19] [2], we have developed novel algorithms and a structure that give very high performance for high dimensionalities.

In A-tree, VBRs can be represented quite compactly, and thus affect the tree configuration both quantitatively and qualitatively. First, since tree nodes can contain a large number of VBR entries, fanout becomes large, which increases search speed. More importantly, we have a free hand in arranging MBRs and VBRs in the tree nodes. Each A-tree node contains an MBR and its children VBRs. Therefore, by fetching an A-tree node, we can obtain the exact position of the parent MBR and the approximate position of its children.

We evaluated the performance of the A-tree using both synthetic and real data. The results demonstrate the effectiveness of the A-tree in high-dimensional searches. The mechanism of the A-tree is remarkably successful, especially for non-uniformly distributed data sets such as real data sets. For both real data sets and synthetic clustered data sets, the A-tree outperforms the SR-tree and the VA-file in all dimensions up to 64 dimensions, the highest dimension examined in our experiments. As far as we know, 64 is the highest dimension of real data used for performance evaluations of high-dimensional access methods with the only exception being the 100-dimensional EigenFace data used in [22].

Additionally, we present a cost model for the A-tree. The experimental results show that the cost model accurately predicts search performance on synthetic and real data sets even for high dimensionalities.

1.4 Outline

The remainder of this paper is organized as follows. Section 2 provides a comparison and analysis of the SR-tree and the VA-file. Based on the analysis, we present the motivation and design principles of A-tree in Sect. 3. Section 4 describes the definitions and algorithms of A-tree. Section 5 presents the results of a performance evaluation of A-tree and conventional access methods. Section 6 concludes the paper.

2 Performance evaluation of SR-tree and VA-file

2.1 Performance evaluation for high-dimensional search

We evaluated the performance of the SR-tree and the VA-file using synthesized and real data. The experiments used three data sets:

Table 1. Maximum number of entry slots in SR-trees

Dimensionality	Non-leaf	Leaf
4	73	227
8	39	120
16	20	62
24	13	41
32	10	31
40	8	25
48	7	21
56	6	18
64	5	15

(1) Uniformly distributed data sets.
Random point sets uniformly distributed in the range [0.1) in each dimension.
(2) Real data sets.
Feature vectors of hue histograms calculated from color images. The images are shot boundaries extracted from video sequences.
(3) Cluster data sets.
For cluster data sets, the number of clusters is 100 in each data set and the center of cluster is distributed uniformly in the range [0.10). In addition, as the number of objects is N, $N/100$ objects are gathered according to a Gaussian distribution around the center of each cluster.

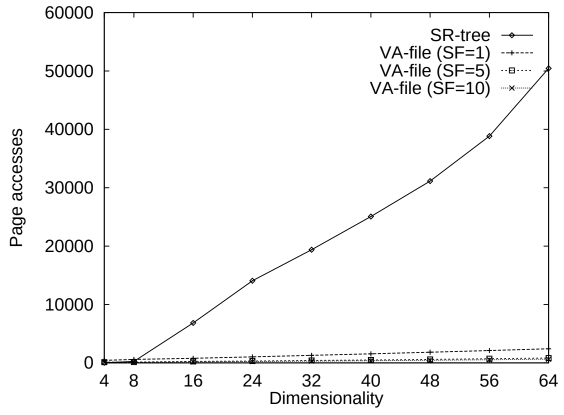
In our evaluation, the dimensionality for synthetic and real data varied from 4 to 64. The size of all data sets was 450,000; page size was 8 kB. In assessing search performance, the page access number and CPU-time were measured by averaging the results of 1,000 queries. The number of nearest neighbor to search was 20 (i.e., $k = 20$). Query points were generated randomly and independently of the data points in indices. CPU-time was measured on a SUN UltraSPARC-II (296 MHz). The search performance of the SR-tree was measured using the algorithm presented in [11] [12], which outperforms the branch-and-bound R-tree traversal algorithm [18] as shown in [3]. The maximum number of entry slots in the SR-tree structures is shown in Table 1.

A VA-file consists of a vector file and an approximation file. Although vector data are accessed randomly, approximation data are visited sequentially in query processing. A sequential scan of approximation data should significantly boost performance because of the sequential nature of their I/O requests. In [24], Weber et al. indicate two speed-up factors for the phenomenon: a practical factor of 10 and a conservative one of 5. In general, the speed-up factor depends on the computing environment. Thus, we used three speed-up factors, $SF = 1$, $SF = 5$, $SF = 10$, in our experimental evaluations. In this paper, the number of page accesses of the VA-file for a speed-up factor SF is given by:

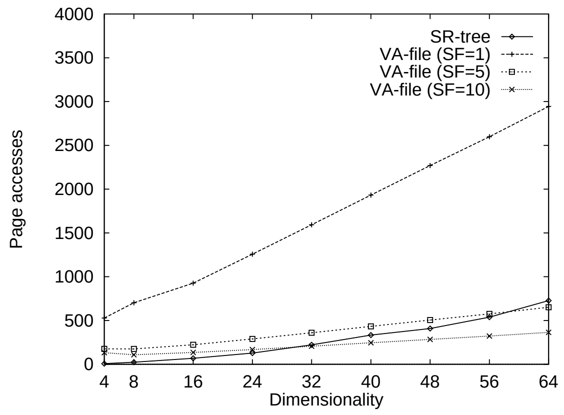
$$PA_{SF} = \frac{PA_{approx}}{SF} + PA_{vector}$$

where PA_{approx} and PA_{vector} show the number of page accesses for approximation files and vector files, respectively.

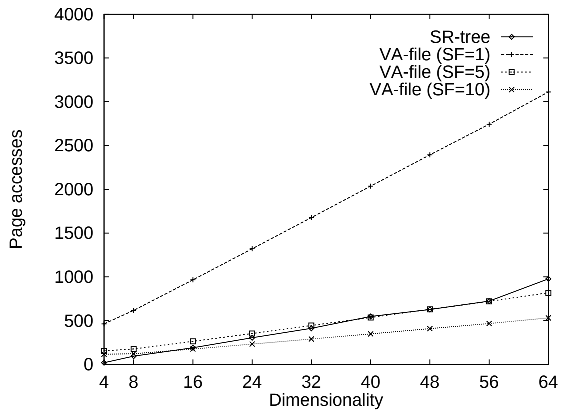
Figure 1 shows the number of page accesses with the VA-file and the SR-tree for synthetic and real data. For uniformly-distributed synthetic data sets (Fig. 1a), the SR-tree is supe-



a



b



c

Fig. 1. Number of page accesses with the VA-file and the SR-tree. **a** Uniformly distributed data sets, **b** real data sets, **c** cluster data sets

rior to the VA-file only for 4-dimensional data. SR-tree performance deteriorates rapidly for data higher than 4 dimensions, and this result agrees with the indication of [24]. For uniformly-distributed data sets, the VA-file is useful, whereas the SR-tree incurs very high search cost. On the other hand, for non-uniformly distributed data sets (Figs. 1b,c), the SR-tree provides better performance compared with the VA-file of SF=1, and it is competitive with the VA-files of SF=5 and SF=10. However, the performance of the SR-tree falls as dimensionality increases. In particular, for real data sets with 64 dimensions, the search cost of the SR-tree is slightly higher than that of the VA-file of SF=5.

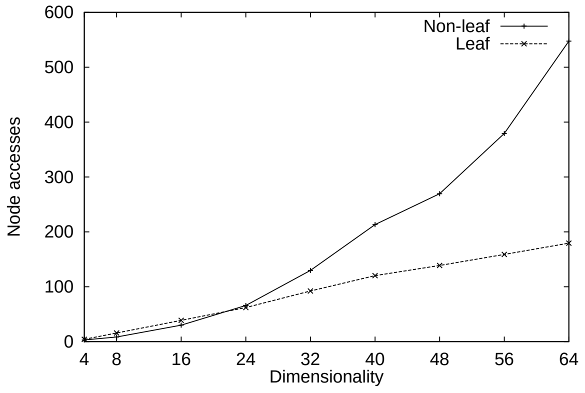


Fig. 2. Number of node accesses with the SR-tree

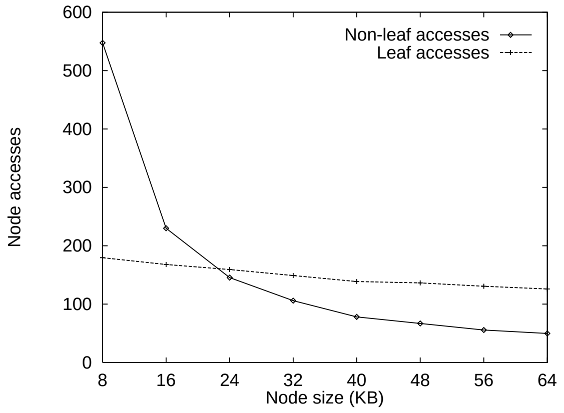


Fig. 3. Number of node accesses versus node size

2.2 Properties of SR-tree

The SR-tree provides sufficient performance in searching real data sets that have non-uniform data distributions. In the SR-tree, areas of dense data points are covered with small MBSs and MBRs. This flexibility effectively increases search performance. Since most multimedia applications generate non-uniformly distributed data sets, the SR-tree seems to be suitable for practical applications. Unfortunately, the search performance of the SR-tree deteriorates with dimensionality. For example, with 64 dimensions the search cost of the SR-tree is slightly higher than that of the VA-file of SF=5. To analyze this phenomenon, we performed more detailed experiments.

Figure 2 shows the number of accesses of non-leaf nodes and leaf nodes for the SR-tree using the real data set under the same conditions as in Fig. 1b. The number of non-leaf node accesses sharply increases with dimensionality. This result can be explained as follows. A non-leaf node consists of entries that contain an MBR and an MBS. MBRs are represented by the coordinate values of two diagonal endpoints, and MBSs are represented by the coordinate values of the center points.

The size of an entry is directly proportional to dimensionality. Thus, the number of entry slots, i.e., the fanout, of a node decreases as dimensionality increases as shown in Table 1. This means that the effectiveness of pruning subtrees diminishes as dimensionality increases. To sum up, the SR-tree incurs a large overhead in handling its non-leaf nodes, and this large overhead increases the number of page accesses.

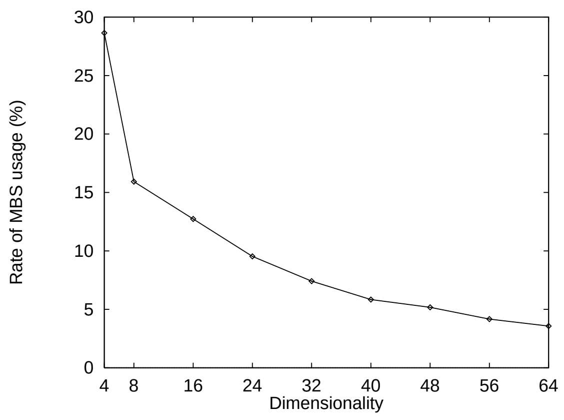


Fig. 4. Rate of MBS usage in node pruning

We measured the number of non-leaf accesses and leaf accesses for SR-trees of various node sizes. Figure 3 shows experimental results for real data sets whose dimensionality is 64. The increase in node page size leads to a larger fanout. As was expected, the number of node accesses decreases as node size increases due to the increase in fanout. Non-leaf accesses were, in particular, considerably reduced. Thus we see that the increase in fanout largely contributes to the reduction in the number of node accesses.

The superior search performance of the SR-tree is due to its strong pruning ability using both MBRs and MBSs. We checked how frequently MBRs and MBSs are used in the search algorithm. Figure 4 shows the ratio of MBS use in searching real data. This figure indicates that the frequency with which MBSs are used decreases as dimensionality increases. The reason for this is that MBSs occupy much larger volumes than MBRs, especially in high-dimensional spaces. Although the center of an MBS is useful for insertion and deletion of data objects, the contribution of MBSs to node pruning is small in high-dimensional spaces.

2.3 Properties of VA-file

In the experiments yielding Fig. 1, following [24], the best code length for the VA-file was chosen from among the three choices: $l = 4$, $l = 6$, and $l = 8$. For real data sets, $l = 8$ for 4–8 dimensions; $l = 6$ for 16 to 64 dimensions. For the cluster data sets, the optimum code length was $l = 6$ for all dimensions. For the uniformly distributed data sets, $l = 6$ for 4 and 8 dimensions and $l = 4$ for 16–64 dimensions.

For uniformly distributed data sets, data objects are stored in cells and are distributed uniformly in those cells. This means that the VA-file suits uniformly distributed data, and it offers high performance as shown in these figures. The VA-file generates short codes since uniformly distributed data objects are pruned efficiently by coarse as well as fine approximation.

With real data or clustered data, on the other hand, many objects may be assigned to one cell since the distribution is non-uniform. The filter step with crude approximation does not work effectively; it follows that non-uniform data sets force the VA-file to choose longer code, which decreases VA-file performance. For instance, Fig. 1a shows that the VA-file (of SF=1) requires almost 2,400 page accesses with uniform data, while Figs. 1b,c show that almost 3,000 page accesses were needed with non-uniform data (both had 64 dimensions).

In the VA-file, filtering by approximation loses its effectiveness with non-uniformly distributed data, so the vector files are visited often. The VA-file does not suit the searching of non-uniform (i.e., practical) data sets since data skew weakens the effectiveness of filtering by approximation file, which consequently degrades the search performance of the VA-file. This problem is critical in practical applications.

2.4 Summary

The evaluation results can be summarized as follows:

(R1) The SR-tree exhibits sufficient search performance for non-uniformly distributed data sets since the tree structure of the SR-tree changes according to the distribution of data sets. In the VA-file, vector data are approximated based on absolute position. Since the approximation of absolute vector positions is independent of data distribution, the VA-file is not suitable for non-uniformly distributed data, commonly found in real applications.
(R2) In the SR-tree, however, like many other indices in the R-tree family, entry size in a node is directly proportional to dimensionality. Hence, as dimensionality increases, fanout becomes small. This degrades search performance.
(R3) Larger fanout contributes to the reduction in the number of node accesses.
(R4) In the SR-tree, the frequency of MBS usage decreases as dimensionality increases. Hence the contribution of MBSs to node pruning is small in high-dimensional spaces.

3 Motivation: introduction of approximation mechanism in tree structures

Section 2.4 summarized our experimental evaluation of the SR-tree and the VA-file. The results revealed that both indices have drawbacks. In this section, we present the design philosophy of the A-tree and show the uniqueness of the A-tree among the proposed indices for high-dimensional data.

3.1 Design philosophy

The A-tree is based on the following design philosophy:

- **Tree structure:** we adopt a tree index based on evaluation result (R1).
- **Relative approximation:** to overcome the problem of tree indices identified in evaluation result (R2), we introduce a new notion, relative approximation, which is a simple yet powerful approximation method utilizing the hierarchy of tree indices. In relative approximation, bounding regions or data points are approximated by their relative positions in terms of the parent's bounding region. Relative approximation has the following benefits:
  – Unlike the absolute approximation offered by the VA-file, approximation values of the relative approximation change in accordance with the data distribution.

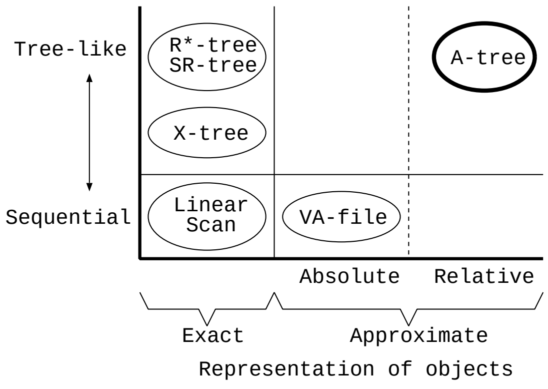


Fig. 5. Classification of spatial access methods

This flexibility makes the approximation error of relative approximation much lower than that of the VA-file. This feature is especially effective for non-uniformly distributed vectors, commonly found in real applications.

– Since approximation values can be compactly represented, entry size in an index node becomes small, which implies larger fanout. This leads to a reduction in the number of node accesses as shown in evaluation result (R3).
– The compact representation of approximated bounding regions allows wider design options of tree configuration freed from the constraints of the traditional tree indices. More concretely, each index node of the A-tree contains a representation of: i) exact position of a bounding region B; and ii) relative approximation of B's children. Therefore, by fetching an A-tree node, we can obtain (partial) information on bounding regions for two generations. This configuration is also useful for efficiently handling update operations.

On the negative side, approximation error may degrade the power of subtree pruning in searching. From the performance evaluation presented in Sect. 5, we have confirmed that the benefits of relative approximation well offset the approximation error.

- **Partial usage of MBSs:** since the SR-tree is one of the best indices among the tree indices proposed so far, the SR-tree is used as the start point in designing the A-tree. However, as shown in the evaluation result (R4), the effect of MBSs is limited when searching high dimensional data. Hence, MBSs are not stored in the A-tree; the centroid of data objects in a subtree is used only for insertion and deletion.

The A-tree is a new index that applies the notion of relative approximation to the hierarchical structure of the SR-tree. However, this application is not naive; A-tree's configuration is unique in that: i) each node contains an MBR and a representation of the relative approximation of its children; and ii) the centroid of data objects is used only for updating purposes.

3.2 Classification of indices

Figure 5 shows a classification of spatial access methods from two viewpoints: representation of spatial objects and index structure. Conventional spatial access methods can be roughly classified into the following three categories: i) linear scan; ii) the VA-file, a sequential file of absolute approximations of feature vectors; and iii) the R-tree family, which has tree structures. The R-tree family can be further classified into "pure" tree-structured indices such as the R*-tree and the SR-tree, and "hybrid" indices having both tree and sequential structures such as the X-tree, which, with the notion of a supernode, shows the stronger property of a sequential scan as dimensionality increases. The A-tree does not belong to any of these categories and is unique in that: i) the A-tree is a tree-structured index; and ii) the representation of MBRs and data objects is based on approximations relative to the parent MBRs.

4 Data structure and algorithms of A-tree

In this section, we first define VBR (virtual bounding rectangle), which is a representation of the relative approximation of an MBR or data object in the A-tree. We then describe the structure of the A-tree. Algorithms for searching and updating are also presented. The nearest neighbor search algorithm is guaranteed to return exact answers, that is, the A-tree gives the desired objects without omission.

4.1 Virtual bounding rectangle

A VBR is a rectangle that contains and approximates an MBR or a data object. In an A-tree, children MBRs and data objects are approximated as VBRs whose positions are relative to their parent MBR.

A rectangle A in n-dimensional space is represented by the two endpoints $\boldsymbol{a}$ and $\boldsymbol{a'}$ of its major diagonal: $A = (\boldsymbol{a}, \boldsymbol{a'})$, where $\boldsymbol{a} = (a_1, a_2, \ldots, a_n)$, $\boldsymbol{a'} = (a'_1, a'_2, \ldots, a'_n)$, and $a_i \leq a'_i$ for $i \in \{1, 2, \ldots, n\}$. Let $B = (\boldsymbol{b}, \boldsymbol{b'})$ be a rectangle contained in A, where $\boldsymbol{b} = (b_1, b_2, \ldots, b_n)$, $\boldsymbol{b'} = (b'_1, b'_2, \ldots, b'_n)$. Hence, $a_i \leq b_i \leq b'_i \leq a'_i \ (i = 1, 2, \ldots, n)$ holds. The basic idea of relative approximation is to quantize the start value b_i and the end value b'_i of the interval (b_i, b'_i) relative to the interval (a_i, a'_i). The quantization functions for the start and end values are similar but slightly different.

We will define the quantization functions more specifically. Let $q \ (\geq 1)$ be an integer. The quantization function Q_s for start values is defined as follows:

$$Q_s(b_i) = a_i + \frac{(a'_i - a_i)h_s(b_i)}{q}$$

where

$$h_s(b_i) = \begin{cases} q-1 & (b_i = a'_i) \\ \left\lfloor \left(\frac{b_i - a_i}{a'_i - a_i}\right) \cdot q \right\rfloor & (\text{otherwise}). \end{cases}$$

Similarly, the quantization function Q_e for end values is defined as follows:

$$Q_e(b'_i) = a_i + \frac{(a'_i - a_i)h_e(b'_i)}{q}$$

where

$$h_e(b'_i) = \begin{cases} 1 & (b'_i = a_i) \\ \left\lceil \left(\frac{b'_i - a_i}{a'_i - a_i}\right) \cdot q \right\rceil & (\text{otherwise}). \end{cases}$$

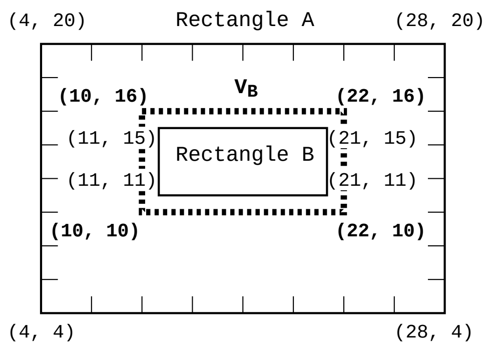


Fig. 6. An example of spatial representation using VBR

Note that $h_s(b_i) \in \{0, 1, \ldots, q-1\}$, and $h_e(b'_i) \in \{1, 2, \ldots, q\}$. In addition, it is easy to verify that the following property holds:

$$a_i \le Q_s(b_i) \le b_i \le b'_i \le Q_e(b'_i) \le a'_i. \tag{1}$$

The virtual bounding rectangle (VBR for short) of B (in A with radix q) is the rectangle

$$V_B = (\boldsymbol{v}, \boldsymbol{v'})$$

where

$$\boldsymbol{v} = (Q_s(b_1), Q_s(b_2), \ldots, Q_s(b_n)),$$
$$\boldsymbol{v'} = (Q_e(b'_1), Q_e(b'_2), \ldots, Q_e(b'_n)).$$

From property (1), B is contained in V_B, and V_B is contained in A. Let C be a point contained in A. Since a point can be regarded as a special rectangle whose two diagonal endpoints coincide, the VBR of C (in A with radix q) can be similarly defined. Figure 6 shows an example of a VBR. In this figure, V_B is the VBR of B in A with radix 8.

The VBR V_B of B (in A with radix q) can be represented by $2n$ integers $h_s(b_i)$, $h_e(b'_i)$ $(i = 1, 2, \ldots, n)$. Since the number of possible values of $h_s(b_i)$ is q, $h_s(b_i)$ can be represented by a binary code of length l $(= \lceil \log_2 q \rceil)$. The same discussion applies to $h_e(b'_i)$. More specifically, we define the binary representation of $h_s(b_i)$ as $[h_s(b_i)]_2$. In addition, we use $[h_e(b'_i) - 1]_2$ as the binary representation of $h_e(b'_i)$. Here, $[x]_2$ is the binary number of integer x. We call the binary code of length $2nl$, which is obtained by concatenating these $2n$ binary codes of length l, the subspace code of V_B (in A with radix q). For the point C, the VBR V_C of C (in A with radix q) can be represented by n integers $h_s(b_i)$ $(i = 1, 2, \ldots, n)$. Hence, the subspace code of V_C (in A with radix q) is of length nl. For example, for the V_B in Fig. 6, $h_s(b_1) = 2$, $h_e(b'_1) = 6$, $h_s(b_2) = 3$, and $h_e(b'_2) = 6$. Hence, 010101011101, which is the concatenation of four binary codes $[2]_2$, $[6-1]_2$, $[3]_2$, $[6-1]_2$ of length 3 (= $\lceil \log_2 8 \rceil$), is the subspace code of V_B in A with radix 8. The subspace code of a VBR V is denoted by $sc(V)$.

Obviously, there is a tradeoff between code length and approximation error. We discuss this tradeoff in Sect. 5.

4.2 Index structure

Besides MBRs and data objects, the subspace codes of VBRs are included in A-tree's structure. Figure 7 shows an example. In Fig. 7a, rectangle R represents the entire space. $M1$ and $M2$ represent rectangles in R. $M1$ is the MBR of $M3$ and $M4$. $M3$ is the MBR of data objects P_1 and P_2. In this structure, $V1$, $V2$, $V3$, and $V4$ are the VBRs of $M1$ in R, $M2$ in R, $M3$ in $M1$, and $M4$ in $M1$, respectively. In addition, $C1$ and $C2$ are the VBRs of $P1$ and $P2$ in $M3$, respectively.

As shown in Fig. 7b, A-trees consist of index nodes and data nodes. Index nodes other than the root in A-trees contain exactly one MBR, say M, and the subspace codes of the VBRs of MBRs in children nodes. M is the MBR of MBRs (or data objects) contained in children nodes. The root node contains no MBR; instead, the entire data space is assumed. When creating A-trees, subspace codes of children VBRs in a node can be calculated from the information of their parent MBR in the same node and the children MBRs (or data objects). In case of the root node, subspace codes of children VBRs are calculated from the information of the entire data space and the children MBRs. Conversely, when searching for data objects, the absolute positions of children VBRs of a node can be calculated from the parent MBR and subspace codes of those VBRs stored in the same node. Therefore, an A-tree node contains partial information of MBRs in two consecutive generations; namely the exact position of an MBR and the approximate positions of its children MBRs. A-tree's search algorithm effectively uses these VBRs for node pruning.

As shown in Fig. 7, A-tree nodes are classified into data nodes and index nodes. Index nodes are further classified into leaf nodes, intermediate nodes, and the root node. The configuration of each type of node is described below:

1. Data nodes:
   a data node has a list of entries (P_1, o_1), (P_2, o_2), ..., (P_m, o_m), where P_i $(i = 1, 2, \ldots, m)$ is the spatial vector of a data object, and o_i $(i = 1, 2, \ldots, m)$ is the pointer to the data object description record. The number of entries in a leaf, m, is bounded by predefined minimum and maximum.
2. Leaf nodes:
   there is a one-to-one correspondence between data nodes and leaf nodes. The leaf node corresponding to a data node $N((P_1, o_1), (P_2, o_2), \ldots, (P_m, o_m))$ has:
   (a) a rectangle M, which is the MBR of $P_1, P_2, \ldots, P_m$;
   (b) a pointer to N; and
   (c) a list of entries $sc(V_1), sc(V_2), \ldots, sc(V_m)$, where V_i is the VBR of P_i in M $(i = 1, 2, \ldots, m)$.
3. Intermediate nodes:
   an intermediate node, i.e., an index node other than root or leaf nodes, contains:
   (a) a rectangle M, which is the MBR of children nodes' MBRs; and
   (b) a list of entries, each of which is a quadruplet $(ptr, sc(V), \omega, P_{centroid})$, where ptr is the pointer to a child node C, V is the VBR of the MBR contained in C, ω is the number of all data objects contained in the subtree rooted by C, and $P_{centroid}$ is the centroid of the data objects in the subtree.

   As explained in Sect. 3.1, MBS radius is not stored in intermediate nodes. ω and $P_{centroid}$ are used only for the insertion or deletion of data objects. Accordingly, in the implementation, an intermediate node is partitioned into two, where the first partition contains the data of M, ptr,

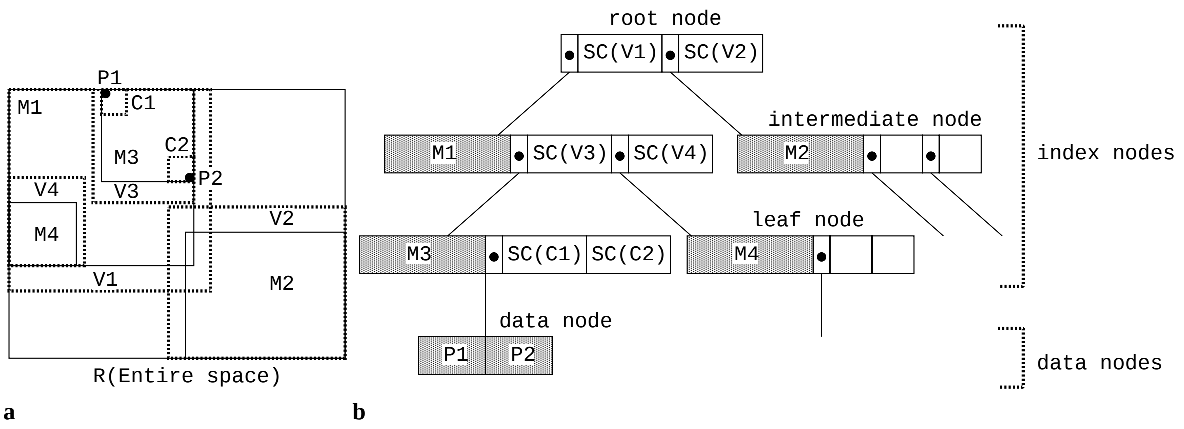


Fig. 7. The A-tree structure. **a** VBR and MBR, **b** tree structure

and $sc(V)$ because the search algorithm uses only M, ptr, and $sc(V)$. The maximum number of entry slots in an intermediate node is determined based on the size of entries in the first partition. Since the first partition is designed to occupy one page, the second partition which contains ω and $P_{centroid}$, occupies more than one page. This method of implementation yields a larger fanout and a faster search time since accesses in searching are limited to the first partitions.

4. The root node:
the root node has entries of the form: (ptr, $sc(V)$, ω, $P_{centroid}$), where ptr is the pointer to a child node C, V is the VBR of the MBR contained in C, ω is the number of all data objects contained in the subtree rooted by C, and $P_{centroid}$ is the centroid of the data objects in the subtree.

4.3 Full utilization

In data-partitioning index trees, the number of entries is less than the maximum number of entry slots in most of the nodes.[1] That is, there are many empty slots in index nodes. We present a new technique, called full utilization, which uses all disk pages in an A-tree structure. With this technique, blank disk space is equitably distributed among all entries in a node. The distributed space is then unevenly assigned to each dimension in an entry. The amount of assigned space depends on the edge length of the parent MBR. Dimensions along which the MBR has a longer edge have higher priority and a large number of bits are assigned to those dimensions. This assignment reduces the approximation error.

Let e_{max} be the maximum number of entry slots in a node. A node has available space of size $l \cdot n \cdot e_{max}$ for all entry slots, where l is the length of subspace code and n is the dimension of the space. Full utilization evenly shares this space among the entries. Hence, if $e(\leq e_{max})$ is the number of entries stored in a node, each entry is assigned a space of length:

$$L_{entry} = \frac{l \cdot n \cdot e_{max}}{e}.$$

[1] The exception is the Hilbert R-tree [13]; this method can utilize almost 100% of the page space.

Note that the code of length L_{entry} is equitably assigned to each entry. If E_i is the edge length of a parent MBR on the ith dimension ($i = 1, \ldots, n$), the code length for approximating the ith position coordinate in an entry is determined as:

$$L_i \approx \frac{L_{entry}}{n} + W, \quad W = \log_2 \frac{E_i}{(\prod_{j=1}^{n} E_j)^{\frac{1}{n}}}$$

where W is the weight of code length that depends on the edge length of a parent MBR. With full utilization, code length is calculated in every node accessed for searching and updating.

4.4 Searching

Figure 8 shows the k-nearest neighbor search algorithm for the A-tree, which is an improvement on the algorithm in [11] [12]. In the algorithm of [11] [12], which is intended for traditional tree structures, the minimum distances (i.e., `MINDIST`) between MBRs in a node and a given query point are calculated and kept in a priority queue. The priority queue is sorted in ascending order of `MINDIST`. Nodes are visited from the top of the queue until the queue becomes empty. In addition, a list is maintained to keep the k-nearest objects found during the execution of the algorithm. The priority queue is pruned by eliminating nodes whose distance to the query exceeds that to the kth nearest neighbor object in the list.

When applying the traditional algorithm to the A-tree, the queue sort incurs a high CPU cost because information about not only the VBRs calculated from MBRs but also the VBRs of data objects are kept in the queue. To process nearest neighbor queries efficiently when a substantial amount of data is stored in the queue, the search algorithm performs filtering using two nearest neighbor lists. One is the usual nearest neighbor list created using the traditional algorithm. Candidate data objects and their distance from the query point are stored in this list. It is called the NNOL (nearest neighbor object list) in this paper. The other list stores the maximum distance from the query point to the VBRs of data objects, and is called the NNVL (nearest neighbor VBR list).

This enhanced search technique is useful only for the A-tree. When searching for the nearest neighbor on the A-tree structure, data stored in the queue can be effectively filtered by using NNVL since the size of the VBRs calculated from

```
Procedure search(point $query$, integer $k$)
 1. enqueue($a\_pointer\_to\_the\_root$, 0);
 2. for $i$ = 1 to $k$, $NNOL[i]$
                := ($node:$ dummy, $dist:\infty$);
 3. for $i$ = 1 to $k$, $NNVL[i]$ := $\infty$;
 4. while emptyQueue() = false do
 5.   $N$ := dequeue();
 6.   if $N$ is a data node then
 7.     for each $entry \in N$ do
 8.       if DIST($query$, $entry$.vector) $\le$
                                  $NNOL[k].dist$ then
 9.         $NNOL[k].node$ := $entry$.oid;
10.         $NNOL[k].dist$ :=
               DIST($query$, $entry$.vector);
11.         sort $NNOL$ by $dist$;
12.         pruneQueue($NNOL[k].dist$);
13.       endif
14.     enddo
15.   else        // $N$ is an index node
16.     for each $entry \in N$ do
17.       $vbr$ := decode($N$.MBR, $entry$.sc(VBR));
18.       if MINDIST($query$, $vbr$)
               $\le NNOL[k].dist$ and
             MINDIST($query$, $vbr$)
               $\le NNVL[k]$ then
19.         enqueue($entry$.ptr,
               MINDIST($query$, $vbr$));
20.         if $N$ is a leaf node and
             MAXDIST($query$, $vbr$)
                  $\le NNVL[k]$ then
21.           $NNVL[k]$ := MAXDIST($query$, $vbr$);
22.           sort $NNVL$;
23.           pruneQueue($NNVL[k]$);
24.         endif
25.       endif
26.     enddo
27.   endif
28. enddo
29. output($NNOL$);    // output the result
```

Fig. 8. k-nearest neighbor search algorithm

data objects is small. On the other hand, this idea is ineffective for MBRs used in traditional tree structures (e.g., X-tree and SR-tree) since the maximum distance between MBRs and a query point is too large to use for making a nearest neighbor list.

In procedure $search$ (see Fig. 8), for initialization, the pair of the pointer to the root and 0 is stored in the priority queue (step 1). In step 5, the function `dequeue()` dequeues this pair from the top of the priority queue, extracts a pointer to a node from the pair, traverses the extracted pointer, and fetches a node. If a fetched node is an index node, the positions of VBRs are calculated from the MBR of the node and the subspace codes for all entries by the function `decode()` (step 17). If the distance between $query$ and a VBR is less than or equal to the kth distance in NNOL (and the kth distance in NNVL), the function `enqueue()` inserts the pair of the pointer to the corresponding child node and the distance into the queue, then sorts the queue in ascending order of distance (steps 18 and 19). In steps 20–24, MAXDIST$(query, vbr)$, the maximum distance from $query$ to each VBR, is calculated. If the distance is less than or equal to the kth distance in NNVL, NNVL is updated and the function `pruneQueue()` is executed. This function reduces the queue size by eliminating pairs in the queue whose distance to $query$ is longer than the argument.

If the extracted node is a data node, data objects in the node are examined (steps 6 and 7). If the distance between $query$ and the data object is less than or equal to the kth nearest neighbor object found so far, the data object together with its distance is stored in NNOL as a nearest neighbor candidate (steps 8–10), and NNOL is sorted (step 11). Furthermore, queue filtering is performed using NNOL (step 12).

We explain the search algorithm using Fig. 7a as an example. First, VBRs $V1$ and $V2$ are calculated from the position coordinates of R, $sc(V1)$ and $sc(V2)$. If the distance from the query point to $V1$ is less than or equal to the distance to the kth nearest neighbor, the node containing $M1$ is fetched, and then VBRs $V3$ and $V4$ are calculated from $M1$, $sc(V3)$ and $sc(V4)$. Similarly, the calculated VBRs are compared with the kth nearest neighbor. If $V3$ is not subject to pruning, the node containing $M3$ is fetched, and then the VBRs $C1$ and $C2$ are calculated from $M3$, $sc(C1)$, and $sc(C2)$, and the calculated VBRs are compared with the kth nearest neighbor. Moreover, MAXDIST from the query point to $C1$ and that to $C2$ are stored in NNVL. If $C1$ is not pruned, $P1$ is accessed.

4.5 Updating

A-tree's update algorithm is based on that of the SR-tree. Starting from data object insertion or deletion, it propagates upward while adjusting MBRs and centroids in non-leaf nodes. The difference between the A-tree and the SR-tree is that the A-tree algorithm needs to calculate and update the subspace codes of VBRs. Concretely, the A-tree structure is updated as follows:

(1) If N_c is a data node, N_p is the parent node of N_c, and M is the MBR of N_c, then M is stored in N_p. If a data object insertion or deletion occurs in N_c, adjust M and the centroid of all data objects contained in N_c.
(2) If M is unchanged, calculate the code of the VBR that approximates the inserted object from M, and update. Otherwise, update the codes of all VBRs stored in N_p.
(3) Let N_c be an index node other than the root node, N_p be the parent node of N_c, and M be the MBR of N_c. If a data object insertion or deletion occurs in the subtree whose top node is N_c, update the centroid of all data objects contained in the subtree, which is stored in N_p. Moreover, if the insertion or deletion causes a change in a child MBR, adjust M.
(4) If M is unchanged, calculate the code of the VBR that approximates the updated child MBR from M, and update. Otherwise, update the codes of all VBRs stored in N_p.

On structures with full utilization, code length for approximating MBRs or data objects in each node varies according to circumstances. Therefore, if the MBR or the number of entries in a node is changed, the codes for all entries in the node must be calculated. Concretely, the A-tree with full utilization performs (2') and (4') instead of (2) and (4):

(2') If M and the number of entries in N_c are unchanged, calculate the code of the VBR that approximates the inserted object, and update. Otherwise, calculate the code length assigned to each dimension for approximating all data objects from M and the number of entries, and update the codes of all VBRs stored in N_p.

(4') If M and the number of entries in N_c are unchanged, calculate the code of the VBR that approximates the updated child MBR, and update. Otherwise, calculate the code length assigned to each dimension for approximating all children MBRs, and update the codes of all VBRs stored in N_p.

4.6 Cost model

This section presents a cost model for the A-tree. The model allows the search cost to be estimated accurately and contributes to finding the optimum code length. It assumes that queries search for the 1-nearest neighbor. A model for k-nearest neighbor queries is also proposed in [14]. Our cost model is based on the number of page accesses since this factor dominates search cost.

4.6.1 Cost model for uniformly distributed data

Before we present the estimation of search cost, we first consider the edge length of VBRs in A-tree structures. Let us assume that an n-dimensional data set whose size is N lies within the n-dimensional unit cube $\mathcal{S} = [0,1]^n$. It follows that the height h of a tree structure can be expressed as:

$$h = \left\lceil \log_{m_{im}} \left(\frac{N}{m_{leaf}} \right) \right\rceil + 1$$

where m_{leaf} is the fanout of a leaf node and m_{im} is that of an intermediate node. Let us assume the tree level of leaf nodes is 1 and the root node is h. Then, on tree level j, the number of nodes N_j is determined as:

$$N_j = \left\lceil \frac{N}{m_{leaf} \cdot m_{im}^{j-1}} \right\rceil \quad (j = 1, 2, \ldots, h). \tag{2}$$

If we assume that $j = 0$ is the data object level, we can regard N_0 as the number of data objects, i.e., $N_0 = N$. Since all MBRs on every tree level fit closely into the entire space, the average volume of MBRs on the tree level j is $1/N_j$. Thus, the edge length of MBRs is given by:

$$t_j = \begin{cases} 0 & (j = 0) \\ \left(\frac{1}{N_j}\right)^{\frac{1}{n}} & (j = 1, 2, \ldots, h) \end{cases}$$

where t_0 is the edge length of data objects. The edge length t'_j of VBRs includes the error $t_{j+1} \cdot 2^{-l}$ for code length l

$$t'_j = \begin{cases} t_j & (j = h) \\ t_j + t_{j+1} \cdot 2^{-l} & (j = 0, 1, \ldots, h-1). \end{cases} \tag{3}$$

Based on [3], the probability that an arbitrary point $\boldsymbol{p}$ must be accessed during a nearest neighbor search for a query point $\boldsymbol{q}$ and expected nearest neighbor distance r is given by :

$$P_{obj}(\boldsymbol{p}) = Vol(SP_n(\boldsymbol{p}, r) \cap \mathcal{S}) = \int_{\boldsymbol{q} \in \mathcal{S}} \left(\begin{cases} 1 & (\|\boldsymbol{q}, \boldsymbol{p}\| \le r) \\ 0 & (\text{otherwise}) \end{cases} \right) d\boldsymbol{q}$$

where $SP_n(\boldsymbol{p}, r)$ is the n-dimensional hyper-sphere whose center point is $\boldsymbol{p}$ and radius is r; Vol denotes volume. r is calculated by the formula described in [3].

Here, on tree level j ($0 \le j \le h$), we assume that an MBR B_j whose edge length is t_j and center point is $\boldsymbol{p} = (p_1, p_2, \ldots p_n)$ lies within the entire space $\mathcal{S}$. Therefore, $\boldsymbol{p}$ lies within the space $\mathcal{S}'_j = [t_j/2, 1 - t_j/2]^n$. The VBR $V_j(\boldsymbol{p}, t'_j)$ that approximates B_j, whose edge length is t'_j, can be expressed using $\boldsymbol{p}$ as:

$$V_j(\boldsymbol{p}, t'_j) = (\boldsymbol{v}, \boldsymbol{v'})$$

where

$$\boldsymbol{v} = (v_1, v_2, \ldots, v_n),\ \boldsymbol{v'} = (v'_1, v'_2, \ldots, v'_n),$$

$$v_i = \begin{cases} 0 & (\text{if } p_i \le \frac{t'_j}{2}) \\ 1 - t'_j & (\text{if } p_i \ge 1 - \frac{t'_j}{2}) \\ p_i - \frac{t'_j}{2} & (\text{otherwise}), \end{cases}$$

$$v'_i = \begin{cases} t'_j & (\text{if } p_i \le \frac{t'_j}{2}) \\ 1 & (\text{if } p_i \ge 1 - \frac{t'_j}{2}) \\ p_i + \frac{t'_j}{2} & (\text{otherwise}). \end{cases}$$

From this, we can obtain the probability that the VBR $V_j(\boldsymbol{p}, t'_j)$ must be visited during the query processing by using Minkowski sum

$$P_{vbr}(V_j(\boldsymbol{p}, t'_j)) = Vol(MS_n(V_j(\boldsymbol{p}, t'_j), r) \cap \mathcal{S}) = \int_{\boldsymbol{q} \in \mathcal{S}} \left(\begin{cases} 1 & (MINDIST(V_j(\boldsymbol{p}, t'_j), \boldsymbol{q}) \le r) \\ 0 & (\text{otherwise}) \end{cases} \right) d\boldsymbol{q} \tag{4}$$

where $MS_n(V_j(\boldsymbol{p}, t'_j), r)$ is the Minkowski sum of $V_j(\boldsymbol{p}, t'_j)$ and r.

As a result, the expected number of page accesses can be obtained as:

$$PA = \sum_{j=0}^{h} \left(N_j \cdot \int_{\boldsymbol{p} \in \mathcal{S}'_j} Vol(MS_n(V_j(\boldsymbol{p}, t'_j), r) \cap \mathcal{S}) d\boldsymbol{p} \right). \tag{5}$$

4.6.2 Cost model for non-uniformly distributed data

Cost estimation for non-uniformly distributed data has also attracted interest. Since real data are typically skewed, the idea of fractal dimension has been used to estimate the intrinsic dimensionality of a data set that is lower than its embedding dimension [4] [7] [3] [14]. However, in the previous works, fractal dimension is not applied to real data of very high dimensionalities in which A-tree is able to work. In this section, we propose a new cost model to estimate the number of page accesses for high-dimensional real data. Although the cost model for non-uniformly distributed data is also based on Eqs. (4) and (5), the edge length of the VBRs and the nearest neighbor distance in non-uniformly distributed data sets depend on the distribution of data sets. The basic idea of our approach is to analyze the distribution of the original data set by using a smaller data set. Based on the analysis, we can obtain the edge length of the VBRs and the nearest neighbor distance.

An outline of our approach can be summarized as follows: first, we create a smaller data set that is selected randomly from the original data set. By using the smaller data set, we determine the distance from a given point $\boldsymbol{p}$ to the ith closest one. Based on this distance, we compute the edge length of the MBR whose center is $\boldsymbol{p}$, which contains m points in the original data set. The edge length of its VBR can be calculated from that of the MBR by using Eq. (3). Similar to the edge length, the nearest neighbor distance that depends on a given query point is also computed by using the smaller data set. By using Eqs. (4) and (5), we finally obtain the expected number of page accesses from the edge length of the VBRs and the nearest neighbor distance.

4.6.3 Edge length of MBRs

Before we estimate the search cost for non-uniformly distributed data, we show the following function for calculating the edge length of an MBR that contains uniformly distributed points. The edge length of the MBR that contains m points uniformly distributed in an n-dimensional unit sphere depends on m and n. The edge length is less than 2 and it increases as m grows. We will apply this function to spheres that contain m points which are close together. We assume that these points are distributed uniformly in the local area, even if they are a part of the entire non-uniformly distributed data set.

Let us assume that m points distributed in a sphere whose center point is the origin O and whose radius is 1, cover a range from $-y_r$ to y_r on a coordinate axis y. A part of the sphere in each of the ranges $-1 \leq y \leq -y_r$ and $y_r \leq y \leq 1$, occupies the volume of $1/2m$ of the whole sphere. Thus,

$$\int_{-1}^{-y_r} S(y)dy = \int_{y_r}^{1} S(y)dy = \frac{1}{2m}\int_{-1}^{1} S(y)dy$$

where $S(y) = Vol(SP_{n-1}(O, \sqrt{1-y^2}))$ is the volume of the hyperplanes of the sphere at the coordinate value y. If we divide the range of $-1 \leq y \leq 1$ into s subintervals ($s \gg m$), the volume of the hyperplanes at $y_i = \frac{2i}{s} - 1$ ($i = 0, 1, \ldots, s$), is $S(y_i)$. Thus, y_r can be obtained as follows:

$$y_r = \frac{2s_r}{s} - 1$$

such that

$$\sum_{i=s_r}^{s} S(y_i) = \frac{1}{2m}\sum_{i=0}^{s} S(y_i).$$

Here, we define the function that gives the edge length of a rectangle that contains m points distributed in an n-dimensional sphere whose radius is d:

$$\Gamma(n, d, m) = 2d \cdot y_r. \tag{6}$$

4.6.4 Cost estimation by analyzing sample data

In our cost model, we analyze the distribution of the original data set by using a smaller data set to obtain: i) the edge length of VBRs which depends on $\boldsymbol{p}$; and ii) the expected nearest neighbor distance which depends on $\boldsymbol{q}$.

Let $\mathcal{D}_N$ be the original data set whose size is N and $\mathcal{D}_\alpha$ be a data set of size α ($\alpha \ll N$) which is selected randomly from $\mathcal{D}_N$. We consider d_i which is the distance between a given point $\boldsymbol{p}$ and the point which is the ith closest to $\boldsymbol{p}$ in $\mathcal{D}_\alpha$. In our method, d_i for every $\boldsymbol{p}$ is calculated from $\mathcal{D}_\alpha$ prior to the cost estimation for $\mathcal{D}_N$. We use d_i for calculating the edge length of MBRs and the nearest neighbor distance in $\mathcal{D}_N$.

In $\mathcal{D}_\alpha$, the sphere whose center point is $\boldsymbol{p}$ and radius is d_{u-1} contains u points including $\boldsymbol{p}$. This sphere contains $N \cdot (u - 1/2)/\alpha$ points in $\mathcal{D}_N$. As shown in Eq. (2), MBRs on tree level j contain $m_{leaf} \cdot m_{im}^{j-1}$ points in a tree structure. We assume that a sphere of which center point is $\boldsymbol{p}$ and radius is d contains exactly these points. Using $\mathcal{D}_\alpha$, d can be computed as:

$$d = \frac{d_{u-1} + d_u}{2}$$

such that

$$u = f_{int}\left(\frac{\alpha \cdot m_{leaf} \cdot m_{im}^{j-1}}{N}\right)$$

where

$$f_{int}(x) = \begin{cases} \lfloor x \rfloor + 1 & (x - \lfloor x \rfloor \leq random(1)) \\ \lfloor x \rfloor & \text{(otherwise)} \end{cases}$$

and $random(1)$ is the function that generates random values whose range is between 0 and 1. The rectangle which contains these points is also given by Eq. (6). Based on this rectangle, we obtain the edge length of the MBRs on tree level j

$$t_j = \begin{cases} 0 & (j = 0) \\ \Gamma(n, d, m_{leaf} \cdot m_{im}^{j-1}) & (j = 1, 2, \ldots, h). \end{cases} \tag{7}$$

The size of VBR can be obtained by using Eq. (3). Since the edge length of MBRs depends on $\boldsymbol{p}$ in our cost model for real data, both t_{j+1} and t_j are calculated based on the same $\boldsymbol{p}$.

Given a query point $\boldsymbol{q}$ and the nearest neighbor distance r_α of $\boldsymbol{q}$ in $\mathcal{D}_\alpha$, the nearest neighbor distance of $\boldsymbol{q}$ in $\mathcal{D}_N$ is estimated by:

$$r_N = r_\alpha \left(\frac{\alpha}{N}\right)^{\frac{1}{n}}. \tag{8}$$

Therefore, the expected number of page accesses for real data can be derived by using Eqs. (5) (7) and (8).

As shown in Sect. 5.1, we have observed that the optimum code length decreases as dimensionality grows. However, a formula for calculating the optimum code length cannot be

Table 2. Maximum number of entry slots in A-trees ($l = 6$)

Dimensionality	Root	Intermediate	Leaf
4	818	812	2706
8	511	503	1342
16	292	283	660
24	204	195	433
32	157	147	319
40	127	117	251
48	107	97	205
56	93	82	173
64	81	71	149

obtained from Eq. (5) since the equations include discrete variables and they are not differentiable. Although the optimum code length depends on data distribution, dimensionality, and set size, the optimum code length shows a narrow range of choices; it ranges only from 4 to 12 under any conditions. It can thus be easily obtained by trying only a few values.

The experiments conducted to show the practical applicability of our cost model are described in Sect. 5.3.

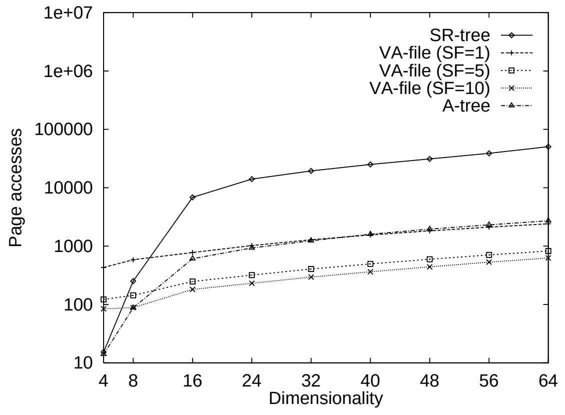
5 Performance test

To verify A-tree's effectiveness, we implemented the algorithm and compared it with the VA-file and the SR-tree. Measurements were made under the same conditions as in the tests in Sect. 2. As for insertion, the average cost for 1,000 insertions was measured; 1,000 objects not included in the indices were inserted into the data sets. For the A-tree, the most superior structure from among five variants, $l = 4$, $l = 6$, $l = 8$, $l = 10$ and $l = 12$, was chosen. The maximum number of entry slots in A-trees with code length $l = 6$ is shown in Table 2 as an example.

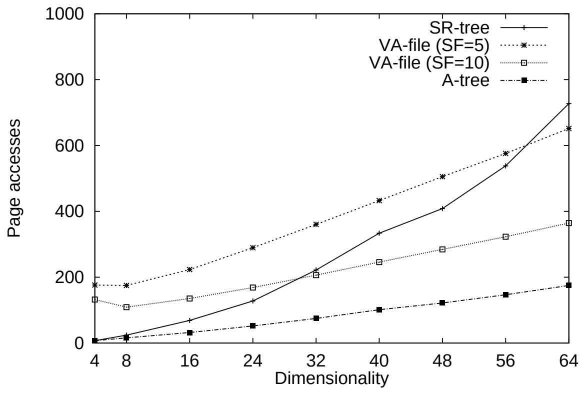
5.1 Search performance

Figure 9 compares the A-tree with the VA-file and the SR-tree. In Figs. 9b,c, the graph of the VA-file (SF=1) is skipped since the search cost is too high. The comparison used code lengths of $l(\in \{4, 6, 8, 10, 12\})$ of the A-tree because these values yield the best search performance for the different levels of dimensionality. The optimum code length for uniformly distributed data sets was $l = 12$ for 4 dimensions, and $l = 4$ for dimensions from 8 to 64. For real data sets, the code lengths selected were $l = 12$ for 4 dimensions, $l = 8$ for 8 dimensions, $l = 6$ for dimensions from 16 to 64. In addition, for clustered data sets, the optimum code length was $l = 12$ for 4 dimensions, and $l = 6$ for dimensions from 8 to 64. We selected the best code length for each dimension. In addition, for the VA-file, the most superior approximation file from among three variants, $l = 4$, $l = 6$ and $l = 8$, was chosen according to [24].

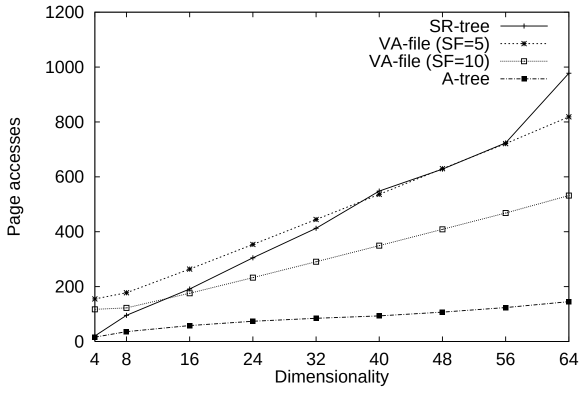
From Fig. 9, in all data sets ranging in dimensionality from 4 to 64, the effectiveness of the A-tree is obvious. The A-tree is almost equal to the VA-file for uniform data sets, and greatly outperforms the other structures for non-uniformly distributed data sets such as real data sets. In particular, the A-tree is remarkably effective for non-uniformly distributed data sets. For example, it needs 75.9% (52.0%) fewer accesses than the SR-tree (the VA-file of SF=10) for real data sets with 64 dimensions. The experimental results in the rest of the paper are for real data sets.



a



b



c

Fig. 9. Number of page accesses versus dimensionality. **a** Uniformly distributed data sets, **b** real data sets, **c** clustered data sets

Figure 10 shows the effectiveness of full utilization. Although both A-trees have the same organization, their coding differs. Since full utilization distributes blank disk space among all entries in a node for coding, VBR approximation error is reduced. Accordingly, this technique decreases the search cost.

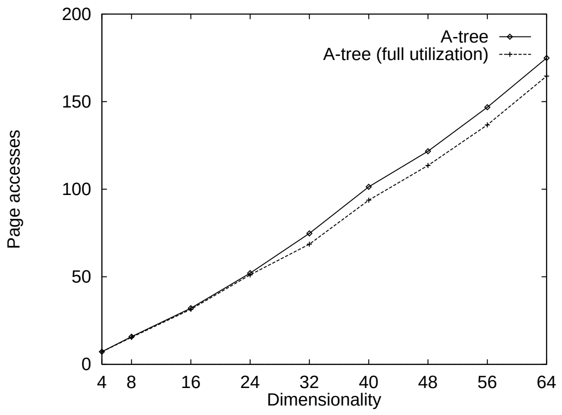


Fig. 10. Number of page accesses for the structure with full utilization

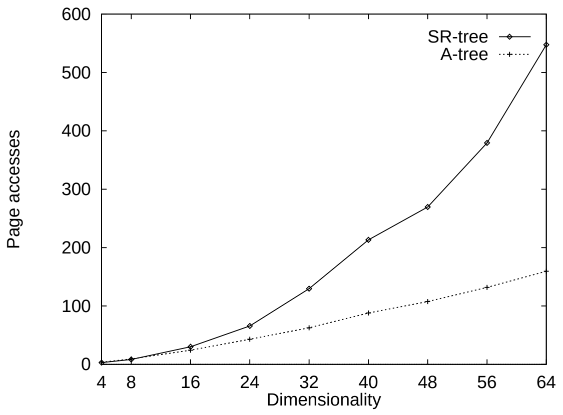


Fig. 11. Number of accesses to index node pages of the A-tree and non-leaf node pages of the SR-tree

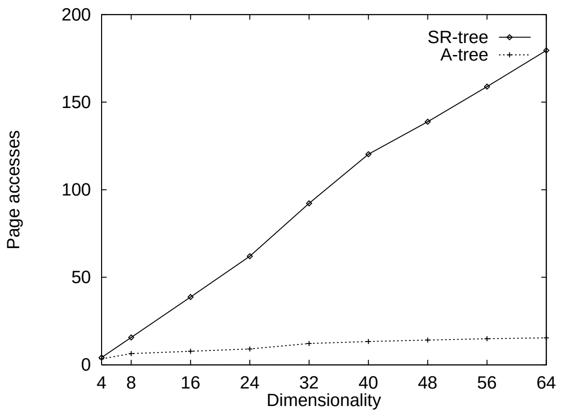


Fig. 12. Number of accesses to data node pages of the A-tree and leaf node pages of the SR-tree

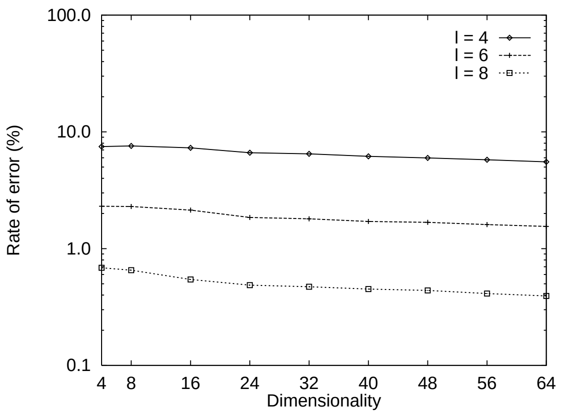


Fig. 13. Approximation error in the distance between query points and VBRs in root and intermediate nodes

5.2 Superiority of A-tree

5.2.1 Comparison of A-tree with SR-tree

Figure 11 shows the number of accesses to index node pages of the A-tree and non-leaf node pages of the SR-tree and Fig. 12 shows the accesses to data node pages of the A-tree and leaf node pages of the SR-tree. The A-tree requires remarkably fewer accesses to both index nodes and data nodes compared with the SR-tree. Moreover, the difference between the two curves increases with dimensionality. Confirming the position stated in Sect. 2.4, one of the most significant problems with the SR-tree is its high search cost for non-leaf node accesses because both MBRs and MBSs are stored in non-leaf nodes. Moreover, this problem also yields a high search cost for leaf node accesses. Since the A-tree is based on relative approximation, the cost of storing VBRs is small. This property increases search performance.

5.2.2 Approximation error with variable length code

VBRs include approximation errors which could degrade A-tree search performance. To assess the tradeoff between approximation error and code length, we measured the approximation error of the distance between query points and visited VBRs in the root and intermediate nodes during a search. Figure 13 plots MBR approximation error against dimensionality for three different code lengths. Note the logarithmic scale of the vertical axis in this figure. We defined the approximation error ϵ of the distance as follows:

$$\epsilon = (1 - r) \cdot 100, \quad r = \frac{1}{S} \sum_{i=1}^{S} \frac{\|\boldsymbol{q}, V_i\|}{\|\boldsymbol{q}, M_i\|}$$

where $\boldsymbol{q}$ is a query point and S is the number of visited VBRs in the root and intermediate nodes during search. V_i are the visited VBRs, and $\|\boldsymbol{q}, V_i\|$ is the distance between $\boldsymbol{q}$ and V_i. M_i is the MBR corresponding to V_i. ϵ was measured by averaging the results of 1,000 queries.

Figure 13 shows the approximation errors versus dimensionality for the A-trees with $l = 4$, $l = 6$ and $l = 8$. This figure shows that the approximation error decreases significantly as code length increases. On the other hand, longer codes cause smaller fanout which might degrade search performance. Hence, there is an optimum code length in terms of search performance. As described in Sect. 5.1, in our experimental setting, the optimum code length changed from $l = 4$ to 12 depending on the dimensionality and distribution of the data sets. In A-trees with optimum code length, the effect of reducing the entry size outweighs the influence of VBR error; consequently, fewer node accesses are required.

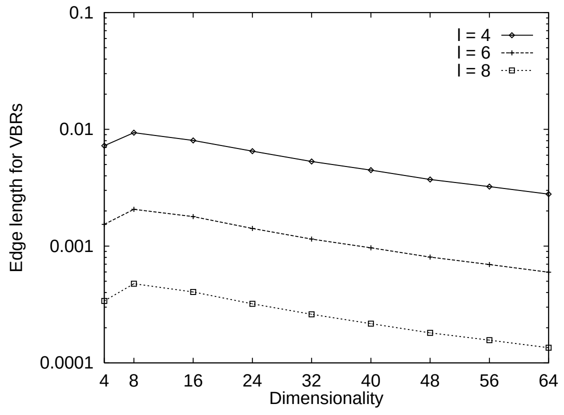


Fig. 14. Edge length of VBRs in leaf nodes

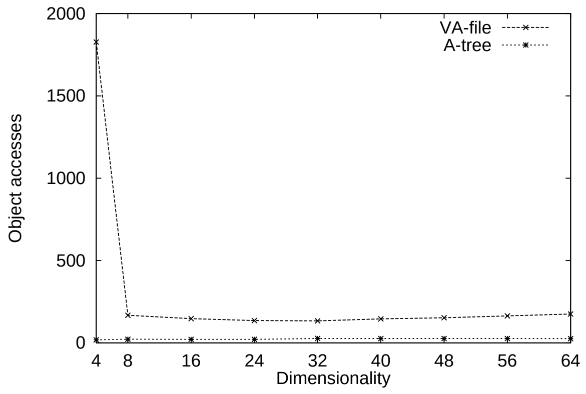


Fig. 15. Number of object accesses versus dimensionality

5.2.3 Comparison of A-tree with VA-file

Although both the A-tree and the VA-file employ the common idea of approximating position coordinates, their data structures and algorithms are completely different. In the A-tree, the approximation is calculated in terms of parent MBR. Therefore, as the level of nodes goes down to the leaves, smaller VBRs are used for approximation. This property is in clear contrast to the approximation computed by the entire space in the VA-file. This difference in data structure causes the difference observed in the accuracy of data object approximation. Figure 14 shows the average edge length of VBRs for data objects. In the VA-file, the edge of each cell occupies the interval 2^{-l}. When compared with the VA-file, the A-tree provides higher accuracy for VBRs as shown in the figure. Consequently, fewer data object accesses are required and search cost is reduced. Figure 15 gives the number of object accesses as a function of dimensionality. As expected, the difference in data object access number between the A-tree and the VA-file is significant.

5.3 Validity of the cost model

To evaluate the accuracy of our cost model described in Sect. 4.6, we compared our cost model and measured A-tree search cost. We show the experimental result based on 1-nearest neighbor queries in this section. For our cost model,

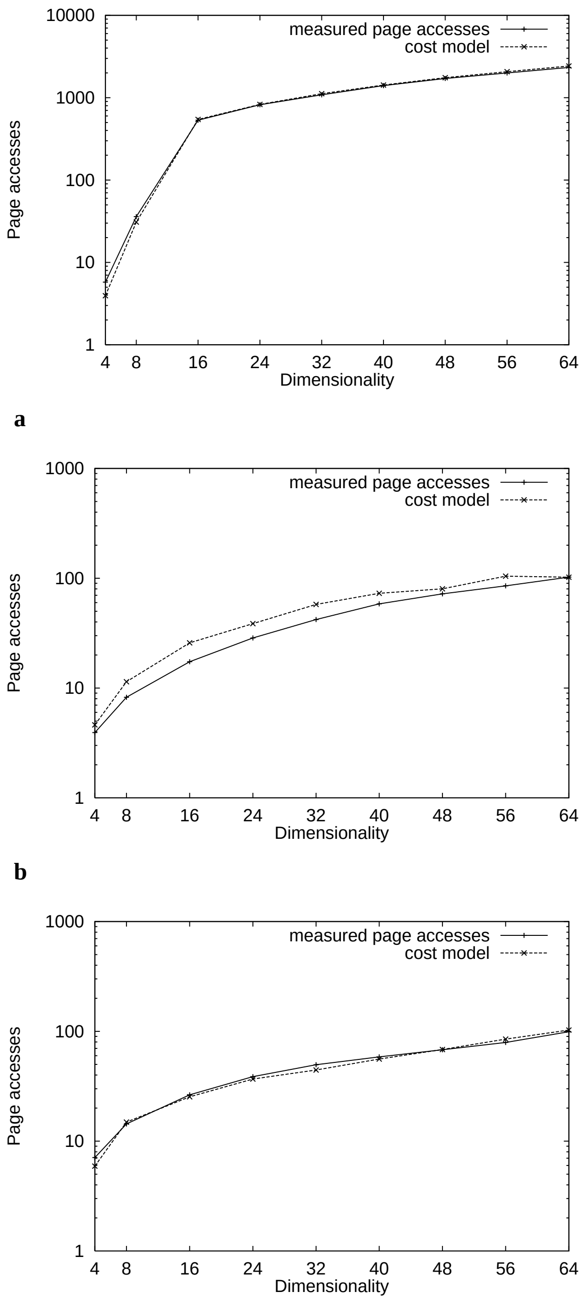


Fig. 16. Expected number of page accesses and measured page accesses versus dimensionality. **a** Uniformly distributed data sets, **b** real data sets, **c** clustered data sets

we assumed that A-tree utilization was fixed at 85%. The experiments used uniformly distributed data sets, real data sets and cluster data sets. The estimation of search cost on the real and cluster data sets was derived by using small data sets whose size is 10,000 (i.e., $\alpha = 10,000$).

For the experiment shown in Fig. 16, we set the size N of the data sets at 450,000 and varied the dimensionality n from 4 to 64. For the experiment shown in Fig. 17, we chose n to be 64 and varied N from $100,000$ to $450,000$.

As shown in these figures, the estimates of our model are sufficiently close to the real performance of the A-tree. Since

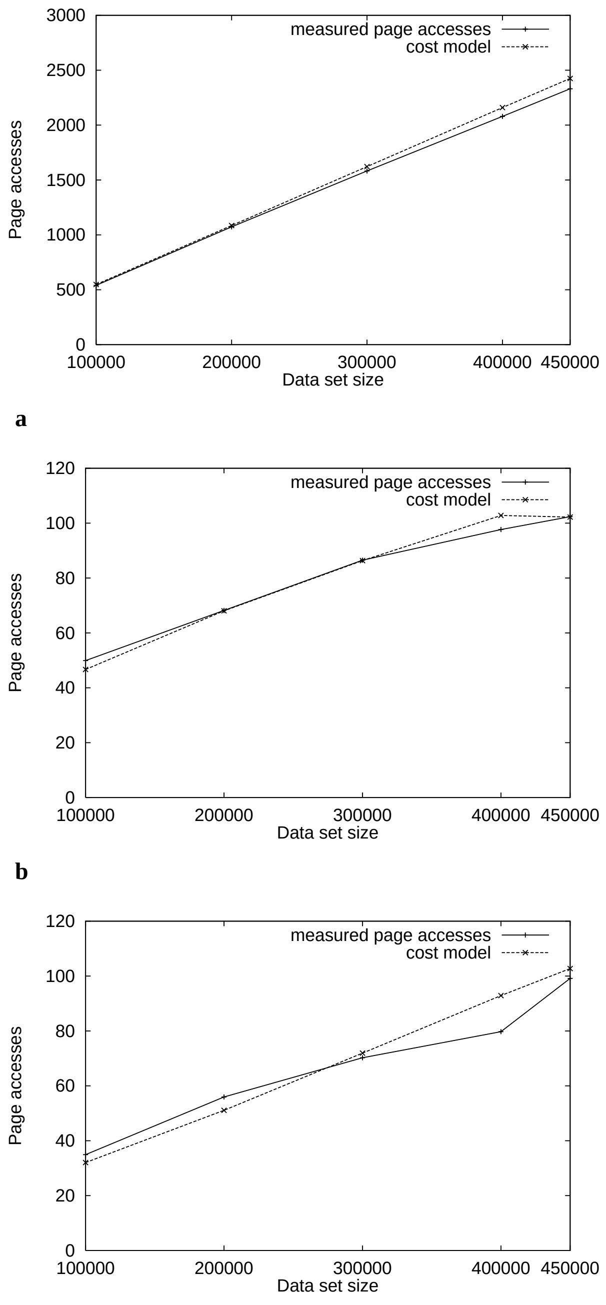


Fig. 17. Expected number of page accesses and measured page accesses versus data set size. **a** Uniformly distributed data sets, **b** real data sets, **c** clustered data sets

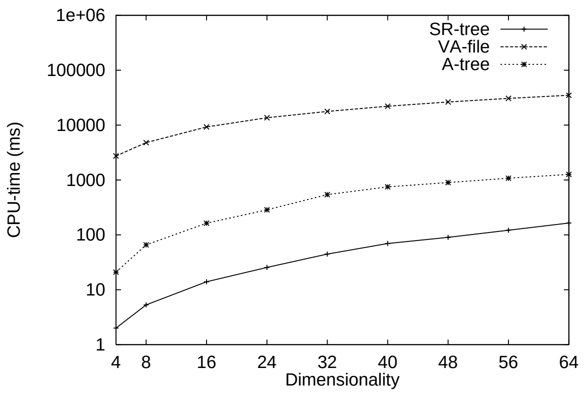


Fig. 18. CPU-time for search

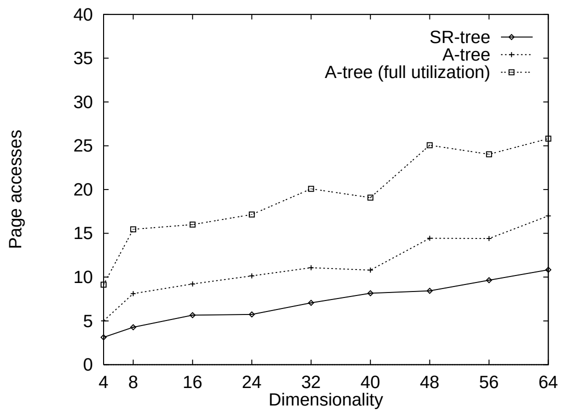


Fig. 19. Page accesses for insertion

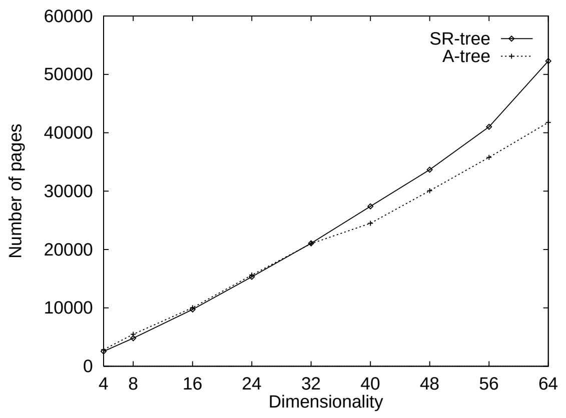


Fig. 20. Storage cost

our model is highly accurate, it is useful in estimating search cost and finding optimal code length for A-tree structures.

5.4 Evaluation of CPU-time

Figure 18 shows the CPU-time measured for the VA-file, SR-tree, and A-tree. CPU-time was measured using the same conditions as in Fig. 9b. The figure indicates that the A-tree has a reasonable CPU cost.

Since the VA-file must calculate the approximate position coordinates for all objects, its CPU-time is much higher than the A-tree's. On the other hand, even though the A-tree needs to calculate VBRs, the CPU-time of the A-tree is lower than that of the VA-file for the following reason. Since the number of node accesses is extremely low, the number of distance calculations is reduced. In addition, the search algorithm filters out the queue using the two nearest neighbor lists. This filtering reduces queue length and provides a further reduction in CPU-time.

5.5 Insertion cost

Figure 19 compares the A-tree with the SR-tree in terms of insertion cost under the same conditions as used in Fig. 9b. Insertion cost was taken as the average cost of inserting 1,000 randomly selected objects. Random objects were used because inserted objects are usually unpredictable in practical situations.

Since the A-tree must access VBRs in addition to MBRs and data objects to maintain the structure, it incurs a larger insertion cost than that of the SR-tree. However, the increase in cost for the A-tree without full utilization is modest. On the other hand, the A-tree with full utilization significantly increases the insertion cost. Since full utilization has a lower search cost, it is suitable for static data sets.

5.6 Storage cost

Figure 20 compares the storage cost of the A-tree with that of the SR-tree under the same conditions as in Fig. 9b. The A-tree and the SR-tree have similar storage costs for 4–32 dimensions, but the A-tree incurs 20.1% less cost for 64 dimensions. The storage cost of the A-tree is low even though it includes VBRs. There are two reasons for this. First, VBRs need only small storage volumes. Second, the number of index nodes in the A-tree is extremely small because of its larger fanout.

6 Conclusions

This paper has presented the A-tree (approximation tree), which offers excellent performance in searching for data in high-dimensional spaces. First, we analyzed the VA-file and the SR-tree, existing structures used for high-dimensional searching, and discussed their problems. Based on this analysis, we developed the A-tree to overcome the problems and achieve higher search performance.

The A-tree's high performance is due to its use of relative approximation. Since the A-tree index nodes contain VBRs, whose storage size is low, the volume of entries in the nodes is reduced. Although VBRs include approximation errors in terms of size, these errors can be reduced by the relative approximation mechanism.

The A-tree is remarkably efficient, especially for non-uniformly distributed data sets such as real data sets. For non-uniformly distributed data sets having up to 64 dimensions, the A-tree outperforms the SR-tree and the VA-file. It requires 75.9% (52.0%) fewer page accesses than the SR-tree (the VA-file of SF=10) for the 64-dimensional real data.

Additionally, we have proposed a cost model for the A-tree. Since the model allows the search cost to be estimated accurately, it is useful in estimating search cost and finding optimal code length for A-tree structures.

Finally, even though VBRs approximate MBRs and data objects, searches yield exact solutions, that is, the A-tree gives the desired objects without omission. The search performance of the A-tree is greatly improved, and its storage cost is low. Thus, the A-tree should be valuable in practical applications.

Acknowledgements. The authors would like to thank Christos Faloutsos (Carnegie Mellon University) for insightful comments on earlier drafts of this paper. We also wish to thank the anonymous reviewers for important suggestions which have been helpful in improving our paper.

References

1. Arya S, Mount DM, Netanyahu NS, Silverman R, Wu AY (1994) An optimal algorithm for approximate nearest neighbor searching. In: Proc. ACM-SIAM Symposium on Discrete Algorithms, pp. 573–582
2. Berchtold S, Böhm C, Jagadish HV, Kriegel HP, Sander J (2000) Independent quantization: an index compression technique for high-dimensional data spaces. In: Proc. IEEE 16th International Conference on Data Engineering, pp. 577–588
3. Berchtold S, Böhm C, Keim DA, Kriegel HP (1997) A cost model for nearest neighbor search in high-dimensional data space. In: Proc. ACM Symposium on Principles of Database Systems, pp. 78–86
4. Belussi A, Faloutsos C (1995) Estimating the selectivity of spatial queries using the 'correlation'. In: Proc. 21st International Conference on Very Large Data Bases (VLDB), pp. 299–310
5. Berchtold S, Keim DA, Kriegel HP (1996) The X-tree: an index structure for high-dimensional data. In: Proc. 22nd International Conference on Very Large Data Bases (VLDB), pp. 28–39
6. Beckmann N, Kriegel HP, Schneider R, Seeger B (1990) The R*-tree: an efficient and robust access method for points and rectangles. In: Proc. ACM SIGMOD International Conference on Management of Data, pp. 322–331
7. Faloutsos C, Gaede V (1996) Analysis of n-dimensional quadtrees using the Hausdorff fractal dimension. In: Proc. 22nd International Conference on Very Large Data Bases (VLDB), pp. 40–50
8. Gaede V, Günther O (1998) Multidimensional access methods. ACM Comput Surv 30(2):170–231
9. Gionis A, Indyk P, Motwani R (1999) Similarity search in high dimensions via hashing. In: Proc. 25th International Conference on Very Large Data Bases (VLDB), pp. 518–529
10. Guttman A (1984) R-Trees: a dynamic index structure for spatial searching. In: Proc. ACM SIGMOD International Conference on Management of Data, pp. 47–57
11. Henrich A (1994) A distance scan algorithm for spatial access structures. In: Proc. ACM International Workshop on Advances in Geographic Information Systems, pp. 136–143
12. Hjaltason GR, Samet H (1995) Ranking in spatial databases. In: Proc. 4th Symposium on Spatial Databases, pp. 83–95
13. Kamel I, Faloutsos C (1994) Hilbert R-tree: An improved R-tree using fractals. In: Proc. 20th International Conference on Very Large Databases, pp. 500–509
14. Korn F, Pagel BU, Faloutsos C (2001) On the "dimensionality curse" and the "self-similarity blessing". IEEE Trans Data Knowl Eng 13(1):96–111
15. Katayama N, Satoh S (1997) The SR-tree: an index structure for high-dimensional nearest neighbor queries. In: Proc. ACM SIGMOD International Conference on Management of Data, pp. 369–380
16. Murase H, Nayar S (1995) Visual learning and recognition of 3-D objects from appearance. Int J Comput Vision 14(1):5–24
17. Pentland A, Moghaddam B, Starner C (1994) View-based and modular eigenspaces for face recognition. In: Proc. IEEE Conf. on CVPR, pp. 84–91

18. Roussopoulos N, Kelley S, Vincent F (1995) Nearest neighbor queries. In: Proc. ACM SIGMOD International Conference on Management of Data, pp. 71–79
19. Seeger B, Kriegel HP (1990) The Buddy tree: an efficient and robust access method for spatial data base systems. In: Proc. 16th International Conference on Very Large Data Bases (VLDB), pp. 590–601
20. Sellis TK, Roussopoulos N, Faloutsos C (1997) Multidimensional access methods: trees have grown everywhere. In: Proc. 23rd International Conference on Very Large Data Bases (VLDB), pp. 13–14
21. Sakurai Y, Yoshikawa M, Uemura S, Kojima H (2000) The A-tree: an index structure for high-dimensional spaces using relative approximation. In: Proc. 26th International Conference on Very Large Data Bases (VLDB), pp. 516–526
22. White DA, Jain R (1996) Similarity Indexing with the SS-tree. In: Proc. IEEE 12th International Conference on Data Engineering, pp. 516–523
23. Wactlar HD, Kanade T, Smith MA, Stevens SM (1996) Intelligent access to digital video: informedia project. IEEE Comput 29(5):46–52
24. Weber R, Schek HJ, Blott S (1998) A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In: Proc. 24th International Conference on Very Large Data Bases (VLDB), pp. 194–205