The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation

Yasushi Sakurai† Masatoshi Yoshikawa§ Shunsuke Uemura§ Haruhiko Kojima‡

† NTT Cyber Space Laboratories, *sakurai@marsh.hil.ntt.co.jp*
§ Nara Institute of Science and Technology, *{yosikawa, uemura}@is.aist-nara.ac.jp*
‡ NTT Cyber Solutions Laboratories, *kojima@aether.hil.ntt.co.jp*

Abstract

We propose a novel index structure, the A-tree (Approximation tree), for the similarity search of 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 rather 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 leads to fast search. More importantly, we have a free hand in arranging MBRs and VBRs in tree nodes. In the A-trees, a node contains an MBR and its children VBRs. Therefore, by fetching a node of an A-tree, we can obtain the information of 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. The A-tree achieves 77.3% (77.7%) fewer page accesses than the SR-tree (the VA-File) for 64-dimensional real data.

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 raise of object recognition ratio, feature vectors of which dimensionality is ten or tens are used in many recognition methods[MN95][PMS94] or systems[WKSS96]. 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[SRF97][GG98] have been proposed so far. This paper introduces a new index structure, named A-tree (Approximation tree) [SYUK00], that offers remarkably higher search performance than existing indices.

1.2 Related Work

The conventional approach to supporting similarity search in high-dimensional vector space 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 so far. They include the R-tree[Gut84], the R*-tree[BKSS90], the Hilbert R-tree[KF94] and the SS-tree[WJ96]. Also, nearest neighbor search methods using such indices have been proposed[RKV95][HS95]. Two recently proposed indices, the X-tree[BKK96] and the SR-tree[KS97], are reported to offer good performance. The X-tree[BKK96] introduces the notion of supernode, and outperforms the R*-tree. The SR-tree[KS97] 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 others, the VA-File (Vector Approximation File)[WSB98] 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 [WSB98], Weber et al. have reported that the VA-File outperforms both the R*-tree and the X-tree when the dimensionality is high ($\geq$ around 6.) To sum up, among access methods for high-dimensional vector space search, the SR-tree and the VA-File are two methods which are not reported to be outperformed by other methods[1] .

In the field of spatial search of high-dimensional data, the well-known problem, the "curse of dimensionality", looms large before us. Of late, search methods that present an approximate answer [AMN+94] [GIM99], have been proposed to avoid the problem. Although these works are useful, the goal of our work is to overcome the problem by developing a search method that gives an exact answer.

1.3 Introduction of the A-tree

This paper proposes a new index structure, the A-tree. 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. Based on the experiments, we have developed a new tree index structure, A-tree, and 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. VBRs can be represented rather 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 leads to fast search. More importantly, we have a free hand in arranging MBRs and VBRs in tree nodes. In A-tree, a node contains an MBR and its children VBRs. Therefore, by fetching a node of an A-tree, we can obtain the exact position of the parent MBR and approximate position of its children.

We evaluate the performance of the A-tree using both synthetic and real data. The results demonstrate the effectiveness of the A-tree in high-dimension search. 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. A-tree achieves 77.3 % (77.7 %) savings in page access compared to the SR-tree (the VA-File) for real data with 64-dimensions. 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 [WJ96].

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 Section 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. Finally, Section 6 concludes the paper.

[1] Recently, Berchtold et al. have reported that the IQ-tree outperforms the VA-File in the range of dimensionality up to 16 [BBJ+00].

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

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

2 Performance Evaluation of the SR-tree and the 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:

(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 extracted from color images.

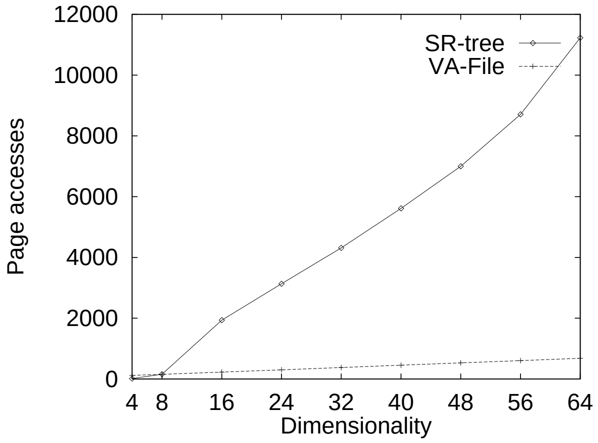
(3) Cluster data sets
For cluster data sets, the number of clusters is 100 in each data set and the center of cluster is distributed uniformly in the range [0.10). In addition, as the number of objects is N, $N/100$ objects are gathered according to Gaussian distribution around the center of cluster.

In our evaluation, the dimensionality for synthetic and real data was varied from 4 to 64. The size of all data sets was 100,000; page size was 8KB. For the A-tree and the SR-tree, one node occupied one page (i.e. 8KB) because both indices give the best search performance with this configuration. In assessing search performance, the page access number and CPU-time were measured by averaging the results of 1,000 queries. Query points were generated randomly and independently of the data points in indices. 20-nearest neighbor queries were used. CPU-time was measured on a SUN UltraSPARC-II (296MHz). The search performance of the SR-tree was measured using the algorithm presented in [HS95], which outperforms the branch-and-bound R-tree traversal algorithm [RKV95] as shown in [BBKK97]. The maximum number of entry slots in the SR-tree structures is shown in Table 1.

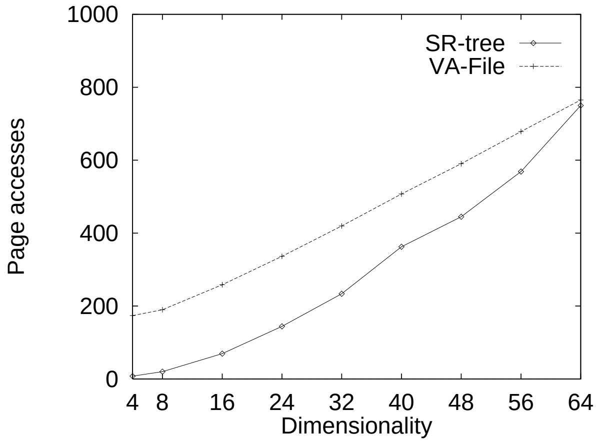
Figure 1 shows the number of page accesses with the VA-File and the SR-tree for synthetic and real data. As shown in Figure 1(a), for uniformly-distributed synthetic data sets, the SR-tree is superior to the VA-File only with 4 and 8 dimensions. SR-tree performance deteriorates rapidly at higher than 8 dimensions and this result agrees with the indication of [WSB98]. For uniformly-distributed data sets, the VA-File is useful, while the SR-tree incurs very high search cost. On the other hand, for non-uniformly distributed data sets, Figures 1(b)(c) show that the SR-tree provides better performance than the VA-File. However, the performance of the SR-tree degrades as dimensionality becomes high. In particular, for real data sets with 64 dimensions, the search cost of the SR-tree is close to that of the VA-File.

2.2 Property of the SR-tree

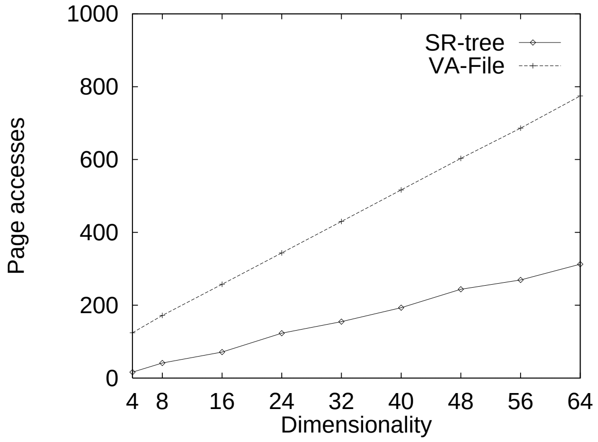
The SR-tree performs better than the VA-File in searching real data sets that have non-uniform data distributions. In the SR-tree, areas of dense data points are covered with small MBSs and MBRs. This flexibility is effective in increasing search performance. Since most multimedia applications generate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



(a)Uniformly distributed data sets



(b)Real data sets



(c)Cluster data sets

Figure 1: Number of page accesses with the VA-File and the SR-tree.

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 used in Figure 1(b). 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 coordinate values of 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, and the number of backtracks 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.

We measured the number of page accesses, non-leaf accesses, and leaf accesses for the SR-tree with various node sizes. Figure 3 shows experimental results for real data sets. The increase in node page size leads to larger fanout. In our experiment, the SR-tree gives the lowest search cost at one page per node; larger node size incurs higher cost. On the other hand, 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 data in real data sets. This figure indicates

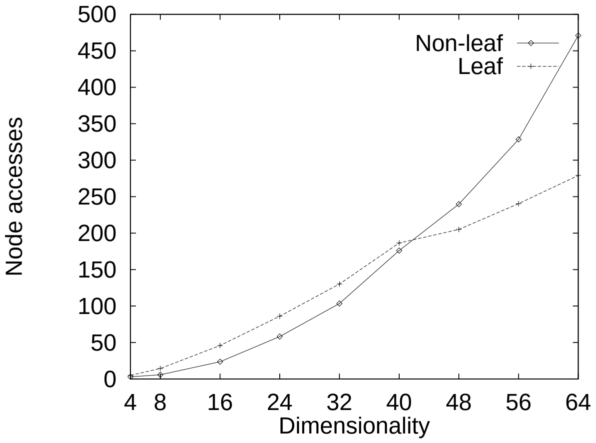


Figure 2: Number of node accesses with the SR-tree.

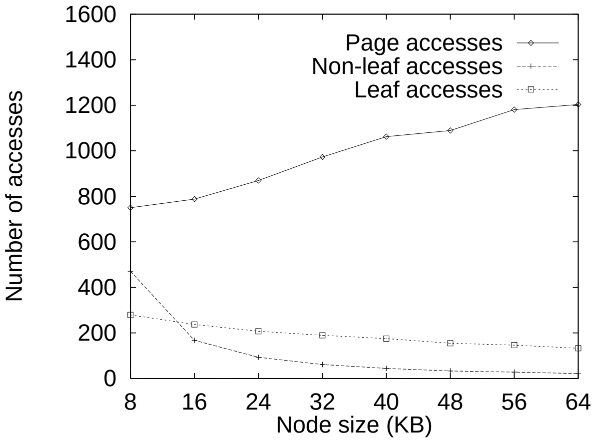


Figure 3: Page and node accesses versus node size.

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 the insertion and deletion of data objects, the contribution of MBSs to node pruning is small in high-dimensional spaces.

2.3 Property of the VA-File

In the experiments yielding Figure 1, following [WSB98], the best approximation file 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 to 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 to 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, Figure 1(a) shows almost 680 page accesses with uniform data, while Figures 1(b)(c) show that almost 770 page accesses were needed with

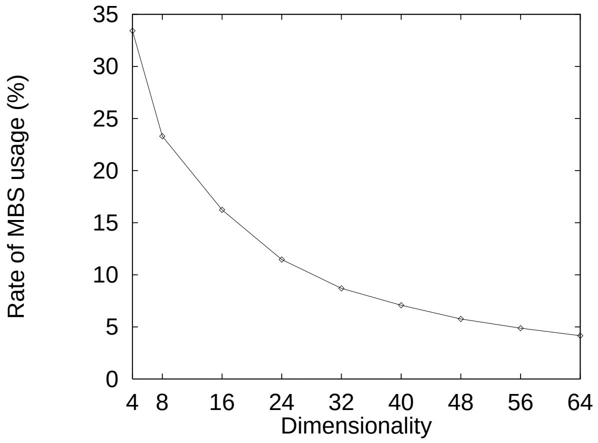


Figure 4: Rate of MBS usage in node pruning.

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

We have performed extensive experiments to analyze the SR-tree and the VA-File. The evaluation results can be summarized as follows:

(R1) For non-uniformly distributed data such as real data and synthetic clustered data, the SR-tree offers better performance than the VA-File. Since the tree structure of the SR-tree changes according to the distribution of data sets, the SR-tree exhibits higher search performance for non-uniformly distributed data sets. In the VA-File, vector data are approximated based on absolute positions. Since the approximation of absolute vector positions is independent of data distribution, many dense data tends to be approximated by the same value. Hence, the absolute approximation leads to large approximation errors for skewed data. Thus, the VA-File is not effective 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 causes the increase of backtracks; thus degrades search performance.

(R3) Increasing node page size leads to an increase in fanout. With real data, the SR-tree provides the lowest search cost at one page per node; the structures for larger node size incur higher cost. However, the larger fanout contributes to the reduction in the number of node accesses.

(R4) In the SR-tree, as dimensionality increases, the frequency of MBS usage decreases since MBSs occupy much larger volumes than MBRs. Hence the contribution of MBSs to node pruning is small in high-dimensional spaces.

3 Motivation: An Introduction of Approximation Mechanism in Tree Structure

Section 2.4 summarized our experimental evaluation of the SR-tree and the VA-File. The results revealed that both indices have their own drawbacks. In this section, based on the summary, 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 The Design Philosophy of the A-tree

The new index data structure, the A-tree, is based on the following design philosophy:

- **Tree Structure:** From evaluation result (R1), we adopt a tree index.
- **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. 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 a node of an A-tree, 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 Section 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; the configuration of the A-tree is unique in that 1) each node contains an MBR and a representation of the relative approximation of its children; and 2) 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: (1) linear scan; (2) the VA-File, a sequential file of absolute approximations of feature vectors; and (3) 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" of tree and sequential structure such as the X-tree, which, with the notion of supernode, shows stronger property of 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

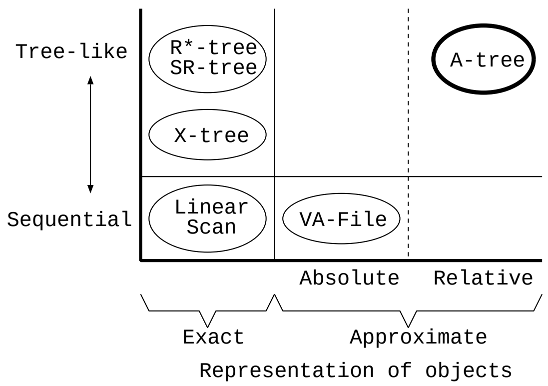


Figure 5: Classification of spatial access methods.

index; and ii) the representation of MBRs and data objects is based on approximations relative to the parent MBRs.

4 The Data Structure and Algorithms of the 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. Also, algorithms for searching and updating are presented. The nearest neighbor search algorithm is guaranteed to return exact answers, that is, the A-tree finds 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 the 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, $a_i \leq b_i \leq b'_i \leq a'_i \quad (i = 1, 2, \ldots, n)$ 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 (≥ 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

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

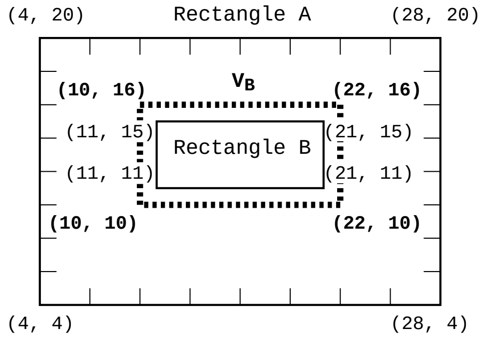


Figure 6: An example of spatial representation using VBR.

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

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

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

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

where

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

From property (1), B is contained in V_B, and V_B is contained in A. Let C be a data object contained in A. Since a data object 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 $h_s(b_i)$, $h_e(b'_i)$ $(i = 1, 2, \dots, n)$. 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 $[h_s(b_i)]_2$. Also, 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 data object C, the VBR V_C of C (in A with radix q) can be represented by n integers $h_s(b_i)$ $(i = 1, 2, \dots, n)$. Hence, the subspace code of V_C (in A with radix q) is of length nl. For example, for the V_B in Figure 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 Section 5.

4.2 Index Structure

Besides MBRs and data objects, the subspace codes of VBRs are included in the structure of the A-tree. Figure 7 shows an example. In Figure 7(a), 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. Also, $C1$ and $C2$ are the VBRs of $P1$ and $P2$ in $M3$, respectively.

As shown in Figure 7(b), 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

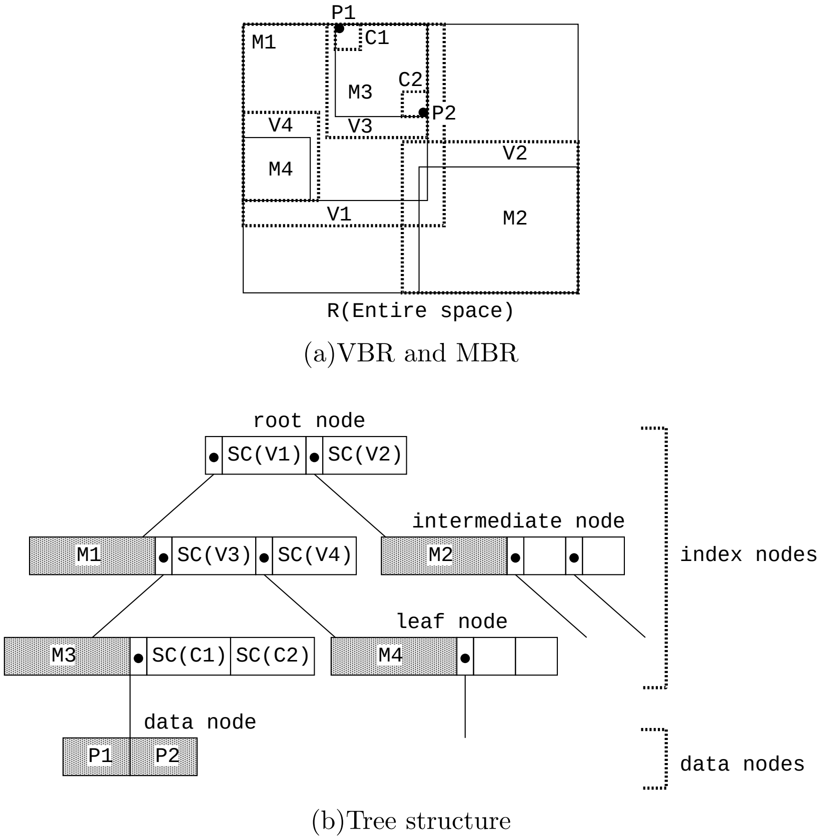


Figure 7: The A-tree structure.

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. Inversely, 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. The search algorithm of the A-tree effectively uses these VBRs for node pruning.

As shown in Figure 7, nodes of the A-tree 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), …, $(P_m, o_m))$ 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 Section 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. The data of ptr and $sc(V)$ are clustered together in the implementation since the search algorithm uses only M, ptr, and $sc(V)$. In an intermediate node, since the first part which contains M, ptr, and $sc(V)$, is designed to occupy one page, the second part which contains ω and $P_{centroid}$, occupies more then one page. This method of implementation yields larger fanout and faster access to the necessary data in searching.

4. The root node:

   The root node has entries of the form: $(ptr, sc(V), \omega, P_{centroid})$, 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.

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.[2] That is, there are many empty slots in index nodes. We present a new technique, **full utilization**, which fully 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 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:

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

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 i-th dimension ($i = 1, \ldots, n$), the code length for approximating the i-th 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.

[2] The exception is the Hilbert R-tree [KF94]; this method can utilize about 100 % of the page space.

```
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 : ∞);
 3. for i = 1 to k, NNVL[i] := ∞;
 4. while emptyQueue() = false do
 5.   N := dequeue();
 6.   if N is a data node then
 7.     for each entry ∈ N do
 8.       if DIST(query, entry.vector) ≤ 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 ∈ N do
17.       vbr := decode(N.MBR, entry.sc(VBR));
18.       if MINDIST(query, vbr) ≤ NNOL[k].dist
           and MINDIST(query, vbr) ≤ NNVL[k] then
19.         enqueue(entry.ptr, MINDIST(query, vbr));
20.         if N is a leaf node and
             MAXDIST(query, vbr) ≤ 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
```

Figure 8: k-nearest neighbor search algorithm.

4.4 Searching

Figure 8 shows the k-nearest neighbor search algorithm for the A-tree, which is an improvement on the algorithm in [HS95]. In the algorithm of [HS95], intended for traditional tree structures, `MINDIST` (i.e. minimum distances) 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. Also, 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 k-th nearest neighbor object in the list.

The priority queue of the search algorithm of the A-tree holds pairs of node pointers and distances. The queue is sorted in ascending order of distance. Since the queue sort incurs high CPU cost if 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 algorithm of [HS95]. 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.)

In Procedure *search* (see Figure 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 k-th distance in NNOL (and the k-th 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 to 24, $\texttt{MAXDIST}(query, vbr)$, the maximum distance from *query* to each VBR, is calculated. If the distance is less than or equal to the k-th 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 k-th nearest neighbor object found so far, the data object together with its distance is stored in NNOL as a nearest neighbor candidate (steps 8, 9, 10), and NNOL is sorted (step 11). Furthermore, queue filtering is performed using NNOL (step 12).

We explain the search algorithm using Figure 7(a) 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 k-th 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 to the k-th 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 to the k-th nearest neighbor. Moreover, `MAXDIST` from the query point to $C1$ and that to $C2$ are stored in NNVL. If $C1$ is not subject to pruning, $P1$ is accessed.

4.5 Updating

The update algorithm of the A-tree 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.

Table 2: Maximum number of entry slots in A-trees.

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

(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.

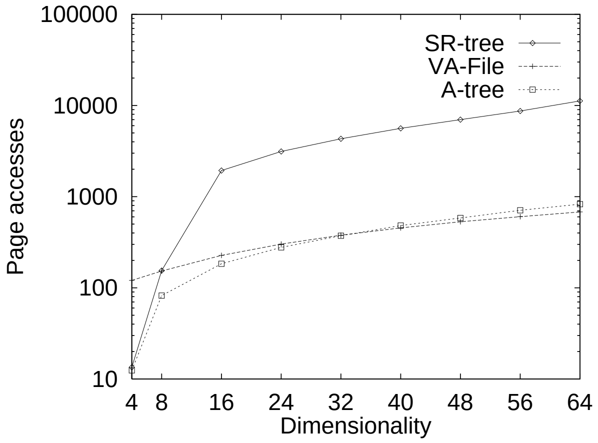
(4') If M and the number of entries in N_c are unchanged, calculate the code of the VBR that approximates the updated child MBR, and update. Otherwise, calculate the code length assigned to each dimension for approximating all children MBRs, and update the codes of all VBRs stored in N_p.

5 Performance Test

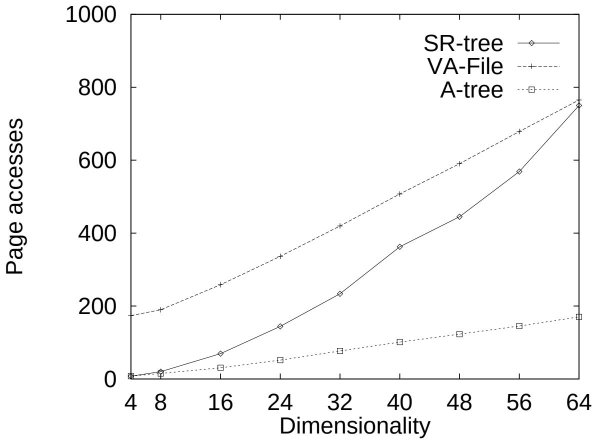
To verify the effectiveness of the A-tree, we implemented the algorithm and compared our proposed method with the VA-File and the SR-tree. The performance of these methods was measured under the same conditions used for the tests in Section 2. As for insertion, the average cost for 1,000 insertions was 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

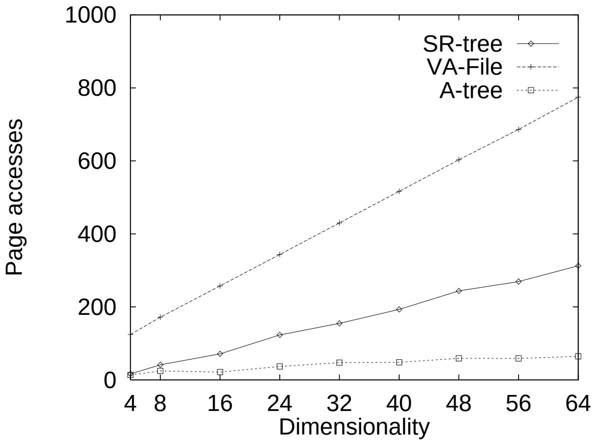
Figure 9 shows a comparison of the A-tree with the VA-File and the SR-tree. The comparison used code lengths of $l(\in \{4, 6, 8, 10, 12\})$ of the A-tree because these values yield the best search performance for the different levels of dimensionality. The optimum code length for uniformly distributed 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. Also, for clustered data sets, the optimum code length was $l = 12$ for 4 dimensions, $l = 6$ for dimensions from 8 to 64. For the comparison shown in Figure 9, we selected the best code length at each dimension. Also, for the VA-File, the most superior approximation file from among three variants, $l = 4$, $l = 6$ and $l = 8$, was chosen according to [WSB98].



(a)Uniformly distributed data sets



(b)Real data sets



(c)Clustered data sets

Figure 9: Number of page accesses versus dimensionality.

As shown in Figure 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. The A-tree is remarkably effective for non-uniformly distributed data sets in particular. For example, the A-tree needs 77.3 % (77.7 %) fewer accesses than the SR-tree (the VA-File) for real data sets with 64 dimensions. The experimental results in the rest of the paper were collected using real 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 A-tree search cost.

5.2 Superiority of the A-tree

5.2.1 Comparison of the A-tree with the 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. Also, Figure 12 shows the accesses to data node pages of the A-tree and leaf node pages of the SR-tree. As shown in these figures, the A-tree

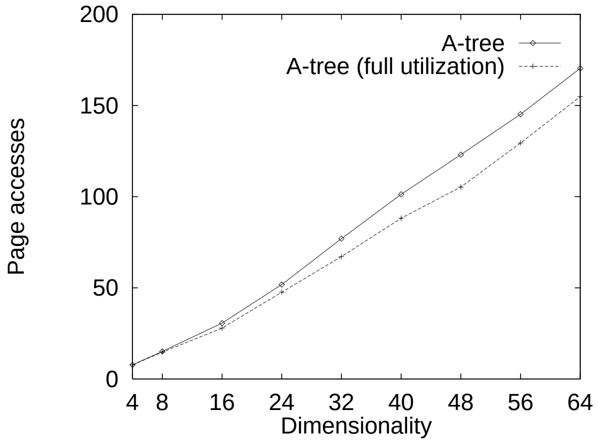


Figure 10: Number of page accesses for the structure with full utilization.

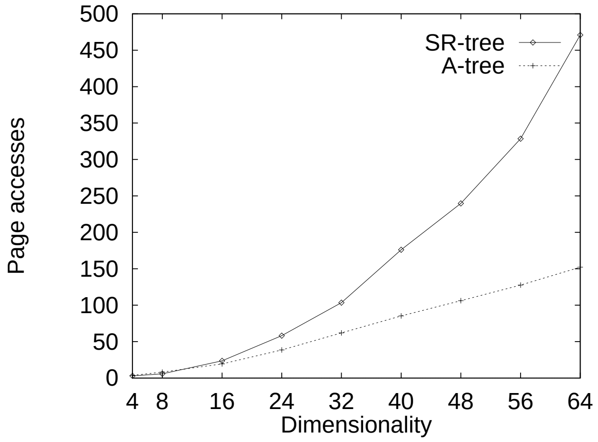


Figure 11: Number of accesses to index node pages of the A-tree and non-leaf node pages of the SR-tree.

requires remarkably fewer accesses to both index nodes and data nodes compared to the SR-tree. Moreover, the difference between the two curves increases with dimensionality. Confirming the position stated in Section 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 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 leads to higher performance.

5.2.2 Approximation Error with Variable Length Code

VBRs include approximation error 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 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:

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

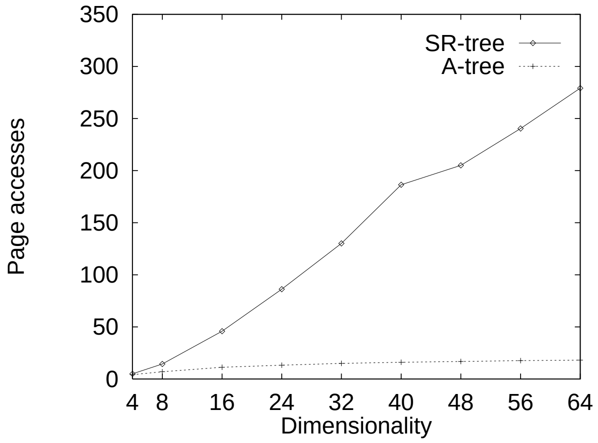


Figure 12: Number of accesses to data node pages of the A-tree and leaf node pages of the SR-tree.

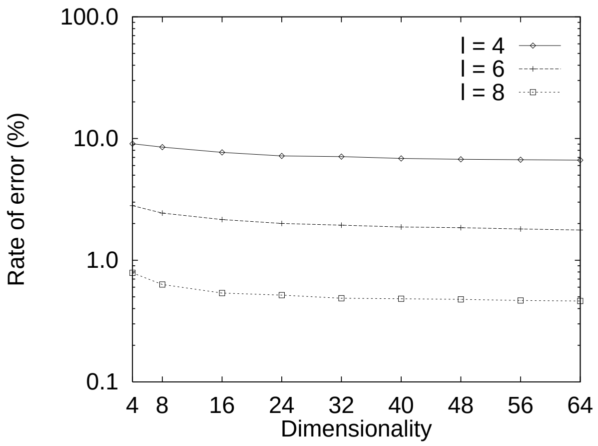


Figure 13: Approximation error in the distance between query points and VBRs in root and intermediate nodes.

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 Section 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.

5.2.3 Comparison of the A-tree with the VA-File

This section explains why the A-tree is superior to the VA-File. Although both the A-tree and the VA-File employ the common idea of approximating position coordinates, their

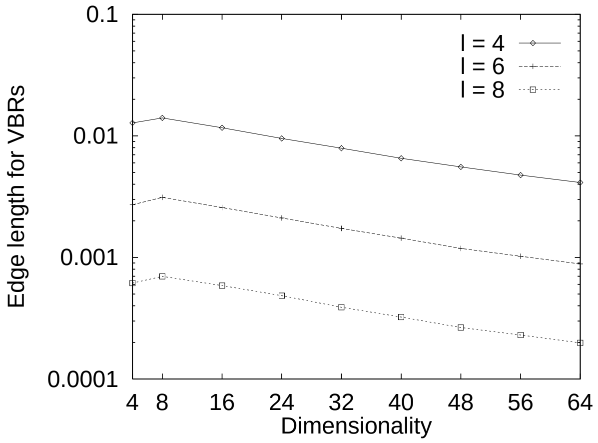


Figure 14: Edge length of VBRs in leaf nodes.

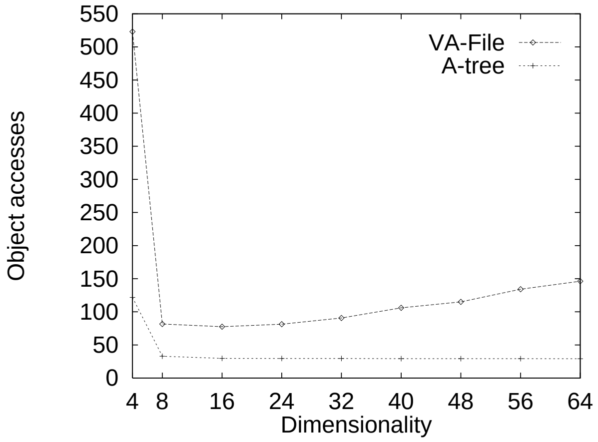


Figure 15: Number of object accesses versus dimensionality.

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 structures 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 to the VA-File, the A-tree provides high 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 Evaluation of CPU-time

Figure 16 shows the CPU-time measured for the VA-File, the SR-tree, and the A-tree. CPU-time was measured using the same conditions used in Figure 9(b). The figure indicates that the A-tree is superior in terms of CPU-time. For 64 dimensions, the performance of the A-tree almost equals to that of the SR-tree and outperforms the VA-File by 81.8 %.

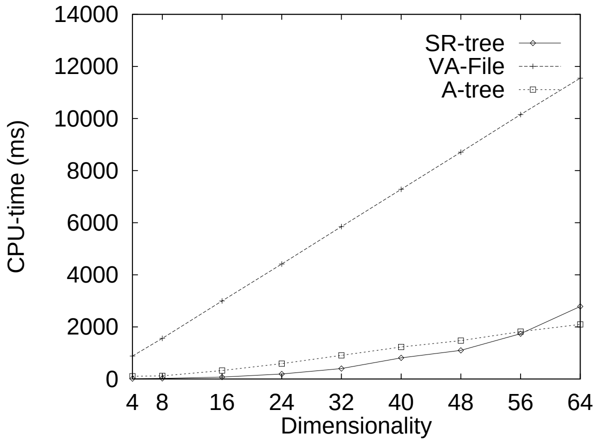


Figure 16: CPU time for search.

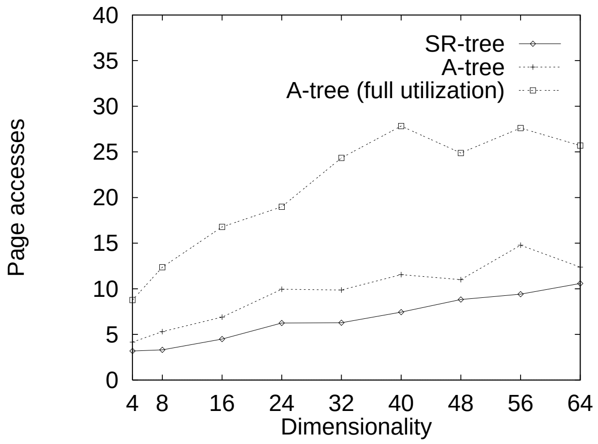


Figure 17: Page accesses for insertion.

Since the VA-File must calculate the approximated position coordinates for all objects, its CPU-time is much higher than the A-tree as shown in Figure 16. On the other hand, although 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 calculation and comparison of the distances from the query point are reduced. Therefore, the A-tree requires less CPU time even though it must calculate VBRs. In addition, the search algorithm filters out the queue using the two nearest neighbor lists which lowers CPU cost. This filtering reduces queue length and provides a valuable contribution to the reduction in CPU-time. As a result, the A-tree provides reasonable CPU cost.

5.4 Insertion Cost

Figure 17 compares the A-tree against the SR-tree in terms of insertion cost under the same conditions as used in Figure 9(b). 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

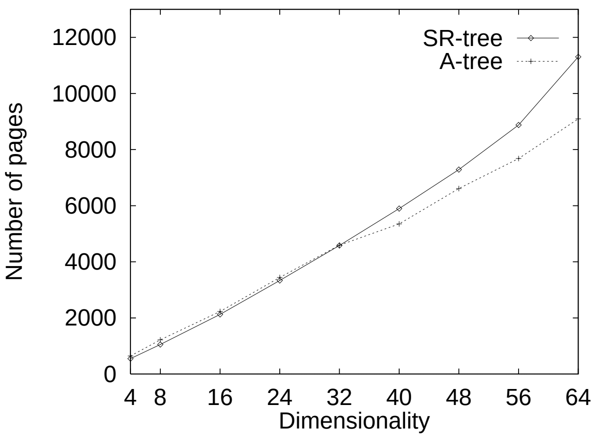


Figure 18: Storage cost.

the structure, the A-tree incurs larger insertion cost than 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 provides lower search cost, this method is suitable for static data sets.

5.5 Storage Cost

Figure 18 compares the storage cost of the A-tree to that of the SR-tree under the same conditions as used in Figure 9(b). The A-tree and the SR-tree incur similar storage costs for 4 to 32 dimensions, but the A-tree incurs 19.5 % 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 due to 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 achieves high performance 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 which yields improved search performance. Although VBRs include approximation error in terms of size, the error is reduced by the mechanism of relative approximation. By using this mechanism, A-tree overcomes the alternatives, the VA-File and the SR-tree.

The mechanism of the A-tree is remarkably efficient, especially for non-uniformly distributed data sets such as real data sets. For non-uniformly distributed data sets, the A-tree outperforms the SR-tree and the VA-File in all dimensions up to 64, which is the highest dimension examined in our experiments. The A-tree requires 77.3 % (77.7 %) fewer page accesses than the SR-tree (the VA-File) for real data with 64-dimensions.

We also present a new technique, full utilization, which fully uses all disk pages in an A-tree structure. This technique provides higher search performance since the approximation errors of VBRs are reduced.

VBRs approximate MBRs and data objects, however, searches yield exact solutions, that is, the A-tree finds the desired objects without omission. The search performance of the A-tree is greatly improved, and its storage cost is low. Thus, this method well supports practical applications.

References

[AMN+94] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu: "An Optimal Algorithm for Approximate Nearest Neighbor Searching", in *Proc. of ACM-SIAM Symposium on Discrete Algorithms*, pp. 573–582, 1994.

[BBJ+00] Stefan Berchtold, Christian Böhm, H. V. Jagadish, Hans-Peter Kriegel, and Jörg Sander: "Independent Quantization: An Index Compression Technique for High-Dimensional Data Spaces", in *Proc. of IEEE 16th International Conference on Data Engineering*, pp. 577–588, 2000.

[BBKK97] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel: "A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space", in *Proc. ACM Symp. on Principles of Database Systems*, pp. 78–86, March 1997.

[BKK96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel: "The X-tree: An Index Structure for High-Dimensional Data", in *Proc. of the 22nd International Conference on Very Large Data Bases (VLDB)*, pp. 28–39, Bombay, September 1996.

[BKSS90] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger: "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", in *Proc. ACM SIGMOD Conf.*, pp. 322–331, Atlantic City, NJ, May 1990.

[GG98] Volker Gaede and Oliver Günther: "Multidimensional Access Methods", *ACM Computing Surveys*, Vol. 30, No. 2, pp. 170–231, June 1998.

[GIM99] Aristides Gionis, Piotr Indyk, and Rajeev Motwani: "Similarity Search in High Dimensions via Hashing", in *Proc. of the 25th International Conference on Very Large Data Bases (VLDB)*, Edinburgh, Scotland, September 1999.

[Gut84] A. Guttman: "R-Trees: A Dynamic Index Structure for Spatial Searching", in *Proc. ACM SIGMOD Conf.*, pp. 47–57, Boston, MA, June 1984, Reprinted in M. Stonebraker, Readings in Database Sys., Morgan Kaufmann, San Mateo, CA, 1988.

[HS95] G. R. Hjaltason and H. Samet: "Ranking in Spatial Databases", in *Proceedings of the 4th Symposium on Spatial Databases*, pp. 83–95, Portland, Maine, August 1995.

[KF94] Ibrahim Kamel and Christos Faloutsos: "Hilbert R-tree: An Improved R-tree using Fractals", in *Proceedings of the Twentieth International Conference on Very Large Databases*, pp. 500–509, Santiago, Chile, 1994.

[KS97] Norio Katayama and Shin'ichi Satoh: "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, May 1997.

[MN95] H. Murase and S. Nayar: “Visual Learning and Recognition of 3-D Objects from Appearance”, *International Journal of Computer Vision*, Vol. 14, No. 1, pp. 5–24, 1995.

[PMS94] A. Pentland, B. Moghaddam, and T. Starner: “View-Based and Modular Eigenspaces for Face Recognition”, in *Proc. IEEE Conf. on CVPR*, pp. 84–91, Seattle, Washington, June 1994.

[RKV95] N. Roussopoulos, S. Kelley, and F. Vincent: “Nearest Neighbor Queries”, in *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 71–79, May 1995.

[SRF97] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos: “Multidimensional Access Methods: Trees Have Grown Everywhere”, in *Proc. of the 23rd International Conference on Very Large Data Bases (VLDB)*, pp. 13—14, Athens, August 1997.

[SYUK00] Yasushi Sakurai, Masatoshi Yoshikawa, Shunsuke Uemura, and Haruhiko Kojima: “The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation”, in *Proc. of the 26th International Conference on Very Large Data Bases (VLDB)*, pp. 516–526, Cairo, September 2000.

[WJ96] David A. White and Ramesh Jain: “Similarity Indexing with the SS-tree”, in *Proc. of IEEE 12th International Conference on Data Engineering*, pp. 516–523, 1996.

[WKSS96] H. D. Wactlar, T. Kanade, M. A. Smith, and S. M. Stevens: “Intelligent Access to Digital Video: Informedia Project”, *IEEE Computer*, Vol. 29, No. 5, pp. 46–52, May 1996.

[WSB98] Roger Weber, Hans-J. Schek, and Stephen Blott: “A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces”, in *Proc. of the 24th International Conference on Very Large Data Bases (VLDB)*, pp. 194–205, New York City, NY, August 1998.