J Supercomput (2012) 62:1362–1384
DOI 10.1007/s11227-012-0800-z

Distributed high dimensional indexing for *k*-NN search

Hyun-Hwa Choi · Mi-Young Lee · Kyu-Chul Lee

Published online: 15 June 2012

Abstract Although conventional index structures provide various nearest-neighbor search algorithms for high-dimensional data, there are additional requirements to increase search performances, as well as to support index scalability for large-scale datasets. To support these requirements, we propose a distributed high-dimensional index structure based on cluster systems, called a Distributed Vector Approximation-tree (DVA-tree), which is a two-level structure consisting of a hybrid spill-tree and Vector Approximation files (VA-files). We also describe the algorithms used for constructing the DVA-tree over multiple machines and performing distributed *k*-nearest neighbors (NN) searches. To evaluate performances of the DVA-tree, we conduct an experimental study using both real and synthetic datasets. The results show that our proposed method has significant performance advantages over existing index structures on different kinds of dataset.

Keywords High-dimensional indexing · Distributed indexing · Approximate *k*-NN query · Cluster system

1 Introduction

Currently, scalability issues in the index mechanism for large-scale data are gaining increasing importance in the fields of cloud computing services, as well as in the re-

H.-H. Choi · M.-Y. Lee
Electronics and Telecommunications Research Institute, Daejeon, Rep. of Korea

H.-H. Choi
e-mail: hyunwha@etri.re.kr

M.-Y. Lee
e-mail: mylee@etri.re.kr

K.-C. Lee (✉)
Chungnam National University, Daejeon, Rep. of Korea
e-mail: kclee@cnu.ac.kr

search domain. In particular, computer applications for content-based information retrieval in the areas of CAD, geography, molecular science, biology, medical imaging, and so on require a large-scale similarity search with feature transformation, where important properties of an object, such as shapes and colors, are mapped into points of a high-dimensional vector space, called feature vectors. A similarity search is defined as a search of points that are close to a given query point in a high-dimensional feature space. In principle, there are two basic types of similarity query: an ϵ-range query and a k-nearest neighbors (k-NN) query. Given a set of data points represented in a high-dimensional space and the distance measurement between them, an ϵ-range query is used to find all objects with a distance smaller or equal to a given range ϵ from a query point, whereas a k-NN query is used to return the k-most similar neighbors of the query point.

To support a similarity search in very large collections of high-dimensional data, a number of distributed index structures for high-dimensional data spaces have been proposed [1–10]. However, most of the approaches recently published focus mainly on supporting range queries [6–8], or operating in peer-to-peer systems [6–10]. In order to provide a content-based information retrieval on massive high-dimensional data in cloud computing services or web search services, we need efficient ways of managing the distributed index that operates in a cluster environment. Meanwhile, the content-based information retrieval is generally based on k-nearest neighbor search for high-dimensional data, because a k-nearest neighbor search unlike an ϵ-range search has no input parameters that require prior knowledge of data. In this paper, we address the k-NN search over large-scale high-dimensional data in cluster environments. A solution is to develop an indexing scheme that can satisfy the following requirements:

- the indexing scheme should be deployable over multiple servers in a cluster environment;
- the indexing scheme should provide a load-balance guarantee on the number of index data stored at each server, regardless of the object distributions in a high-dimensional feature space;
- the indexing scheme should concentrate the index data for similar objects on a small number of servers, in order to make the number of servers accessed for k-NN query executions as small as possible and to improve the parallelism of supporting multiple independent queries;
- the indexing scheme should provide a parallelizable algorithm for efficient k-NN queries on high-dimensional data. In addition, the indexing scheme should guarantee good accuracy for all queries, which can be a problem particularly in the domains with skewed data distributions;
- for newly inserted massive data, the distributed infrastructure should be extendable to other servers.

We propose a high-dimensional indexing structure, called the Distributed Vector Approximation-tree (DVA-tree), which operates in a cluster environment. We have designed a centralized indexing structure to solve the search performance and accuracy problem of decentralized indexing methods. A decentralized index mechanism has high network traffics and a long response delay by the many node accesses and message propagations used to process similarity queries. Locality Sensitive Hashing (LSH) [22] has received considerable attention recently, because it was shown that its runtime is independent of the dimension of feature vectors for querying. However, LSH, which probabilistically assigns similar data to the same bucket in a hash table, is not suitable for skewed data and can be highly inaccurate in a similarity search. The DVA-tree is based on a binary tree structure, which partitions the search space. The DVA-tree is inspired by the idea of a distributed hybrid spill-tree, which partitions a feature vector space in a hierarchical tree manner and then maps leaf nodes of the tree into separate servers in a cluster environment. However, we minimize the overall time required to build an index and retrieve neighbors by using the Vector Approximation-file (VA-file) [11], instead of a hybrid spill-tree [18], as the local index on separate servers. The VA-file, which was developed to accelerate a sequential scan over the entire feature vectors, provides better search performance than the tree-based methods on a single server if the dimensionality of the feature vectors is larger than around 5 dimensions [4]. In conclusion, we propose a mechanism that efficiently distributes VA-files in order to increase search performance, as well as to support index scalability for large-scale data.

With this work, we make the following contributions: (i) We present a new distributed index scheme that combines a tree-based index structure with a scan method using the VA-file; (ii) we present an approximate k algorithm to process the distributed k-NN queries and propose cost formulas for predicting the response time of the k-NN search; (iii) finally, we experiment and evaluate the efficiency and effectiveness of our approach using the skewed data and uniformly distributed data.

The rest of this paper is organized as follows. After reviewing related work in Sect. 2, we provide background information about the hybrid spill-tree and the VA-file in Sect. 3. Section 4 introduces algorithms for creating the distributed index and processing the k-NN search. Section 5 provides detailed evaluations of the proposed experiments. We conclude the paper in Sect. 6.

2 Related work

Over the years, techniques for solving the exact and approximate k-nearest neighbor problems have evolved. For efficient query processing, the partition-based approach, which organizes data in a tree structure, was the first indexing scheme for high-dimensional data [13–19]. The partition-based approach can be divided into two categories: space partitioning methods such as K-D-B tree [13] and hB-tree [14], and data partitioning methods such as R*-tree [15], X-tree [16], M-tree [17], and hybrid spill-tree [18]. The space partitioning methods divide the data space along predefined hyper planes regardless of data distribution. The resulting regions, which are subsets of the data space, are mutually disjointed, with their union taking up the complete space. On the other hand, the data partitioning methods partition a data space according to the distribution of the data. These can yield possible overlapping regions. Therein, the partition-based index structures usually use minimum bounding rectangles (MBRs), or minimum bounding spheres (MBSs) as regions. The difference between the two kinds of partition-based index structure is that the space-partition based indices are usually formed from top to bottom, whereas data-partition based indices are usually formed from bottom to top. The partition-based approach performs very well when data dimensionality is not high. However, with increasing dimension, the processing time grows exponentially, due to a serious overlap of directory nodes in the tree. In very high dimensions of $d \geq 25$, virtually all regions are accessed [19].

In recognition of this fact, an approximation-based approach that performs a sequential scan over the entire data set was developed. Roger et al. introduced the VA-file [11] in order to accelerate the sequential scan by use of the compressed and quantized representations of the feature vectors with bit encodings. The LPC-file [20] and CBF scheme [21] were proposed to significantly enhance the discriminatory power of approximation.

On the other hand, there are many applications of the nearest neighbor search where an approximate answer is applicable. This observation underlines the hash-based approaches [22, 23]. The basic idea behind the hash-based approaches is to hash the objects using several hash functions so as to ensure that, for each function, the probability of collision is much higher for objects which are close to each other than for those which are far apart. At query time, the k-NN search is performed by hashing the query object to one bucket per hash table and retrieving objects stored in buckets containing that object. The LSH scheme efficiently supports the approximate k-NN search, and scales well even for a relatively large number of data dimensions. However, in order to achieve a high level of the search accuracy, multiple hash tables need to be constructed. The number of hash tables required can reach up to over a hundred [22]. Therefore, with limited storage space, the LSH scheme cannot guarantee a good accuracy for all queries which can be a problem especially in domains with non-uniform, high-correlated distributions. For the accuracy and speed guarantees to hold, some of the LSH parameters which need to be adjusted are: the number of hash tables to use, the number of buckets in each table, and the number of hash functions per each table. The difficulty of setting these LSH parameters is one of the deficiencies of the LSH scheme.

The previous methods mentioned above are all designed to be executed on a single server. The traditional algorithms simply cannot be applied to large-scale high-dimensional data that unable to be processed with a single server. In other words, we need efficient ways of distributing the storage and search of the index data across multiple servers.

A number of P2P approaches have been proposed for large-scale similarity searches. Mayank et al. [9] and Parisa et al. [10] leverage the LSH-based approaches to provide approximate results to a k-NN search. However, the hash function makes it difficult to efficiently support a k-NN search, because there is no way to expand the set when the buckets examined do not have enough neighbors. SkipIndex [6] is based on a k-d-tree in partitioning the data space, and maps the data space into a skip graph overlay network by encoding the space into a unique key. The DiST [7] and VBI-tree [8] are also tree-based approaches, which do not scale well when data dimensions are high.

Several approaches to parallelize an NN search on the cluster systems have been developed. The approaches like the Master R-trees [1], Master-Client R-trees [2], and distributed hybrid spill-tree [3] have tried to parallelize the tree-based structures. Roger et al. [4] and Jaewoo et al. [5] proposed a parallel NN-search based on the VA-file to achieve a linear increase of search speed as the number of servers grows. However, the query response times of these solutions are not satisfactory for a search engine which enables contents-based information retrieval on the World-Wide Web.

3 Background

3.1 The hybrid spill-tree

The hybrid spill-tree [17] was introduced to support efficient approximate k-NN searches recently. The hybrid spill-tree is a combination of a metric tree with high search accuracy and a spill-tree with fast search. The difference between the metric tree and the spill-tree is that the children of the spill-tree can share data points, whereas those of the metric tree do not.

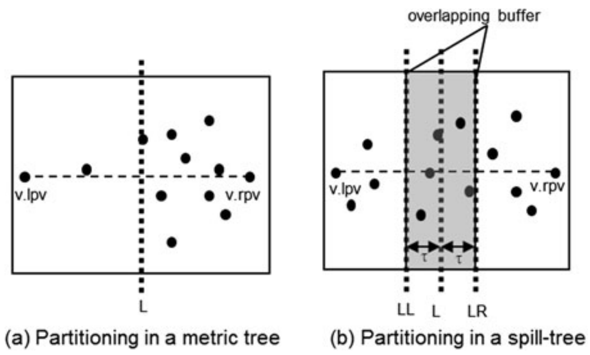
We first explain how to partition an internal node v. See Fig. 1 as an example. Formally, we use $N(v)$ to denote the set of points covered by node v, and use *v.lc* and *v.rc* to denote the left child and the right child of the node v. We choose two pivot points from $N(v)$, denoted as *v.lpv* and *v.rpv*. Ideally, *v.lpv* and *v.rpv* are chosen so that the distance between them is the largest of all pair distances within $N(v)$. We then find the decision boundary L that goes through the midpoint between the pivot points. The partition procedure of a metric tree implies the points sets of *v.lc* and *v.rc* are disjoint, separated by the decision boundary L, as shown in Fig. 1(a). The relationship between two child nodes of the node v in a metric tree is

$$N(v.lc) \cup N(v.rc) = N(v) \qquad (1)$$

$$N(v.lc) \cap N(v.rc) = \Phi \qquad (2)$$

In a spill tree, the splitting strategy allows overlaps between the two children. As illustrated in Fig. 1(b), we define two new separating planes, *LL* and *LR*, both of which are parallel to L and at distance τ from L. Then all the objects to the right of plane LL belong to the child *v.rc*, and all the points to the left of plane *LR* belong to the child *v.lc*. The region between *LL* and *LR* is called an overlapping buffer, because

Fig. 1 Strategies of partitioning


v.lpv
v.rpv
L
(a) Partitioning in a metric tree
overlapping buffer
v.lpv
v.rpv
τ
τ
LL
L
LR
(b) Partitioning in a spill-tree

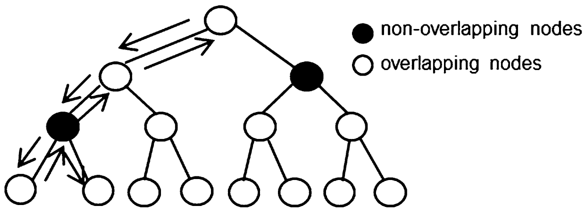


Fig. 2 An overview of the hybrid spill-tree structure

all objects that fall in the region are shared by *v.lc* and *v.rc*. Mathematically, we have

$$\{N(v.lc) \cup N(v.LR)\} = N(v.lc) \tag{3}$$

$$\{N(v.rc) \cup N(v.LL)\} = N(v.rc) \tag{4}$$

A metric tree efficiently finds neighbors on the low-dimensional datasets using a depth first search. However, the metric tree slows down, as the dimension of the datasets increases. Most time on a metric tree search is spent by backtracking to prove a candidate point is the true NN. Based on this observation, a spill-tree was designed to avoid the cost of exact NN verification. A defeatist search on a spill-tree descends the tree using the decision boundaries at each level without backtracking, and then output the point in the first leaf node it visits as the NN of query q. The problem with the defeatist search is on its low accuracy, but the spill-tree defeatist search overcomes the problem by using the overlapping buffer.

In order to achieve a balance between efficiency and accuracy in a search, a hybrid spill-tree is used. As shown in Fig. 2, a hybrid spill-tree has both overlapping nodes and non-overlapping nodes. In the hybrid spill-tree, if the children of a node contain shared points, the node is an overlapping node; otherwise, a non-overlapping node. The key to building a hybrid spill is to determine whether to do the overlapping splitting or not. For each node v, we first split the points using the overlapping buffer. If either of its children contains more than pre-defined fraction value ($\rho < 1$, which is usually set to 0.7) of the total points in node v, we undo the overlapping splitting. Instead, a conventional metric tree partition without an overlapping buffer is used, we mark the node v as a non-overlapping node. In contrast, all other nodes are marked as overlapping nodes. An NN search on a hybrid spill-tree becomes a hybrid of the metric tree depth first search (MT-DFS) for the non-overlapping nodes and the defeatist search for the overlapping nodes.

3.2 The VA-file

The main idea of the VA-file [11] is to compress the feature vectors to reduce the amount of data that must be read during search. The VA-file consists of a vector file which contains vectors and an approximation file which contains a compressed representation of each vector. The approximations are based on a grid that divides the data space into hyper-rectangular cells, each of which can be represented as a bit-string of a user-specified length.

Figure 3 illustrates the vector approximation in a two dimensional vector space. Both the vector file and approximation file for the vector data are unsorted; however, the ordering of the points in the two files is identical. We used $b = 2$ bit per dimension.

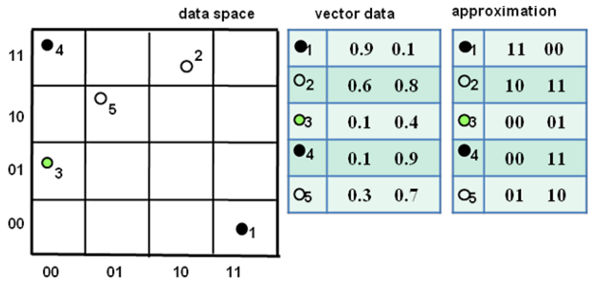


Fig. 3 Structure of the VA-file

Thus, the data space in each dimension is split into $2^b = 4$ partitions. The address of a cell is the concatenation of its partition numbers in each dimension. Finally, each vector is approximated by the bit-string of the cell that it falls. For instance, vector 1 falls in the bottom right cell, which is represented by the bit-string 1100 (11 from the dimension 1 and 00 from the dimension 2).

An NN search can be performed using two phases. In the first phase, the entire approximation file is sequentially scanned, and the upper and lower bounds of the distance between each vector and a query point are computed. The details about a bound computation are described in [12]. A vector is a candidate whenever less than k vectors have been encountered, or whenever the lower bound is less than the kth largest upper bound currently in the candidate set. The first phase eliminates the majority of vectors and returns a small set of the candidate vectors. The second phase refines these candidates by measuring the actual distance to the query point.

4 The DVA-tree

The performance of a k-NN query degrades as the size of the index increases. Therefore, when the amount of index data is very large, it is important to distribute the index data across multiple servers in order to improve the data access performance through the benefit of parallel processing.

In this section, we mainly concentrate on developing a new index structure and algorithm that facilitate a fast approximate k-NN search in cluster environments based on a global file system.

4.1 The DVA-tree construction

The research challenge that has led to the design of the DVA-tree is how to efficiently distribute a large-sized VA-file across multiple servers in cluster environments. This is achieved by clustering the feature vector data. The clustering method is an optimized approach for approximate k-NN queries, as it can significantly reduce the size of the accessed dataset in a search. We propose using the hybrid spill-tree as a clustering method, because the hybrid spill-tree is the latest data partition method that is efficient in both accuracy and time of retrieval.

As shown in Fig. 4, the DVA-tree is a distributed version of a two-level index scheme. The global index of the top level is a hybrid spill-tree, and the local index

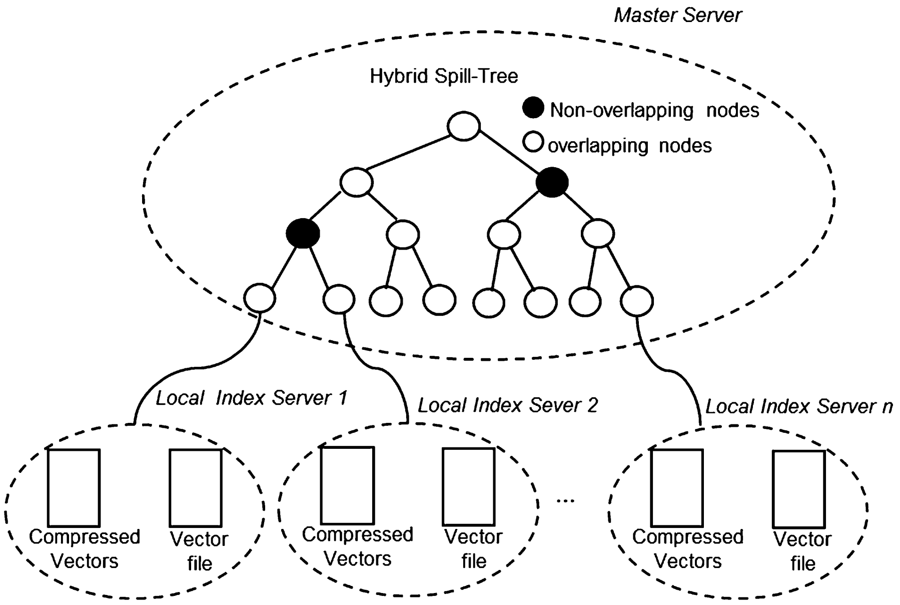


Fig. 4 Structure of the DVA-tree

on separate servers is a VA-file. To search the index, a global index is initially used to determine which local index or indices must be accessed. Since each leaf node in a hybrid spill-tree has the range information of the relevant local index, as well as the name of the server with the local index, it is possible for the master server with the global index to return the names of servers with local index data for a query. We may expect that the master server for the global index in a centralized scheme could create a performance bottleneck, and thus it is necessary to alleviate the overhead on the overloaded master server using various data management facilities such as data replications, third-party copies, data look-up services and etc. The application and implementation of a specific data management facility is not the scope of this paper, as we mainly focus on the architecture and *k*-NN search algorithm for the DVA tree. All *k*-NN queries are forwarded to the determined local index servers that are responsible for processing *k*-NN queries on their own local index data. Herein, some of the local index data may be stored on multiple local index servers, according to the data sharing characteristic of nodes in the hybrid spill-tree.

For constructing a DVA-tree, we first pick a subset of the feature vector data, as the volume of the feature vector data is too large to be accommodated by a single server. To accurately predict clusters of the entire feature vector data, we use the subset obtained from random sampling among the feature vector data. The random sampling method is generally used to build a miniature version of the index structure for large-scale high-dimensional data. The random sampling method is simple and independent of the dimensionality. Furthermore, it captures data distribution effectively, and thus can be used to model uniform or non-uniform data. We create a random sample that is small enough to fit on a single server, because it is difficult for a tree structure to be operated on multiple servers and traversed in parallel. We then build a hybrid spill-tree on the random sample. Since the hybrid spill-tree is built upon the subset of the vector dataset, we have to reduce the page capacity of all leaf nodes in a hybrid spill-tree accordingly. For example, if we use as a sample $1/10$ of the original vector data, each page of the hybrid spill-tree will contain on average $1/10$ of the original points. In other words, the page capacity is also reduced by a factor of 10.

The precision of a hybrid spill-tree depends on the sampling rate. Clearly, a large sample dataset increases the precision of the tree in terms of cluster prediction. On the other hand, too small samples will cause more errors on predicting clusters. As the sample size affects the number of leaf nodes, as well as the height of the hybrid spill-tree, it is difficult to determine an appropriate sampling size for a given dataset. However, we can estimate the minimum sample size with the formula proposed by Yamane [24], regardless of the data distribution:

$$n \geq \frac{N}{N \cdot e^2 + 1} \tag{5}$$

where n is the number of sample vectors, N is the number of indexed vectors, and e is the amount of random sampling error. Finally, we use sample vectors of the number which should be greater than the estimated sample value n. On the other hand, the size of sample vectors must not exceed the total memory size available in a master server, because we assume that the hybrid spill-tree are loaded into the main memory on a master server.

In order to speed up the k-NN query on a VA-file, we also need to consider the case that approximation files should fit into main memory on the local index servers. Herein, the memory size of the local index server should be considered as a parameter for the page capacity within the hybrid spill-tree. For new data, we can typically set the page capacity of the tree as a scale factor of $1/2$ or less than the expected number of objects that will fall into a single leaf node of the created hybrid spill-tree.

Figure 5 shows the parallel algorithm for constructing a DVA-tree in the cluster environments based on a global file system. At first, we pick a subset of the given vector data using a random sampling method. We build a hybrid spill-tree on the sample using the standard tree-building algorithm as described in [18]. We bind each leaf

```
Algorithm M_buildTree (vectorData, servers)
1. sample = sampling(vectorData);
2. tree = buildHybridSpillTree(sample);
3. bind(treeLeafNodes, servers);
4. partitions = partitionVectorSet(vectorData, n); // n is the number of distributors
5. call distributeVectors(tree, partition) in n-parallel distributors;

distributeVectors (tree, partition)
1. for each vector v ∈ partition
2.   serverList = searchHybridSpillTree(tree, v);
3.   for local index server s ∈ serverList
4.     call L_insertVector(v);

Algorithm L_insertVector(v)
1. approxList.addApprox(getVectorApproximation(v));
2. writeVector(vectorFile, v);
```

Fig. 5 The algorithm used in building a DVA-tree

node in the hybrid spill-tree to a separate local index server. In order to distribute the vector data to multiple local index servers, we partition the vector data into a set of n splits. The splits can be processed in parallel by the n different distributors, i.e., processors. A distributor, which is assigned a split, reads the vectors of the corresponding split. The distributor determines the local index servers to insert each vector v using the standard hybrid spill tree search, and forwards the vector v to the designated local index servers. The local index server adds a compressed version of an input vector v to an approximation list in the main memory in order to reduce the amount of disk I/Os during a similarity query process. Then, the local index server stores the input vector v on a vector file.

4.2 The k-NN query algorithm

The overall DVA-tree can be viewed conceptually as a single hybrid spill-tree, spanning a large number of servers. The DVA-tree nodes can be partitioned into three classes: routing nodes, server routing nodes, and data nodes. The routing nodes are all the internal nodes of a hybrid spill-tree on a master server. The routing nodes are used to determine which leaf nodes in the tree must be accessed. The server routing nodes are the leaf nodes in the hybrid spill-tree. Each server routing node points toward an associated data node. The data nodes are separate local index servers containing a bit-compressed version of the points and their exact representation. The data nodes process the k-NN queries based on the approximation-based approach, i.e., the VA-file.

As shown in Fig. 6, the k-NN queries are processed in the three phases. In the first phase, the k-NN query is submitted to the master server that contains the hybrid spill-tree. The master server traverses the hybrid spill-tree in order to determine which VA-file(s) must be accessed. At this time, the master server transforms the k-NN query into a range query with arbitrary threshold, and performs the range query on the hybrid spill-tree. In particular, the master server uses, as threshold for the range query, the average kth distance between the sample data that are computed, while

```
Algorithm KNNSearch (q, k)
1. serverList =M_TreeSearch(q, k);
   // n is the serverList size
2. call candidateSets.addSet(L_VASearch(q, k)) in n-parallel query requestors;
3. knn= merge(candidateSets, k);
4. return knn;

Algorithm M_TreeSearch (q, k)
1. rangeQuery = generateRangeQuery(q, k);
2. serverList = rangeSearch(tree, rangeQuery);
3. return serverList;

Algorithm L_VASearch (q, k)
1. candidates = pruneVectors(approxList, q, k);
2. knn = refineVectors (vectorFile, candidates, q, k);
   // k neighbors in the knn are listed in order of the distances from the query point q.
3. return knn;
```

Fig. 6 k-NN search algorithm for the DVA-tree building the hybrid spill-tree for the sample data. The range search reduces the search costs of the hybrid spill-tree, because it can be executed without backtracking for the non-overlap nodes. The master server returns a list of the n local index servers as the result of the range query, which generates a requirement to perform the k-NN query on the n local index servers. The simplest implementation is to execute n query requestors (processors) in parallel, i.e. a query requestor is executed for each of the local index servers. This is a reasonable choice for a case when a small number of local index servers are expected to be selected. In the second phase, the k-NN query is forwarded to the n local index servers, which are determined by the master server, in different query requestors. The local index servers process the k-NN queries on the VA-files in parallel. A local index server first scans only the approximations in a main memory, and computes an upper and a lower bound on distance between each vector and the given query point q. While doing so, the local index server eliminates the vast majority of the vectors, and finds a small set of vectors with the smallest bounds, so-called the candidates. The remaining vectors are accessed in the vector file to identify the k nearest neighbors, and random IO operations occur. The local index server returns a list of k vectors, as the nearest neighbors, in an increasing real distance order from the query point q. Finally, the results obtained from the n local index servers are merged. The number of the neighbors obtained from the multiple local index servers is $k \cdot n$. The final k nearest neighbors from the $k \cdot n$ vectors are determined by comparing the exact distances between the given query point and $k \cdot n$ vectors.

We illustrate the k-NN search process on the DVA-tree in Fig. 7. When an NN query with a point q and an integer $k \geq 1$ is submitted to the master sever, the master server transforms the NN query into a range query with an average kth distance

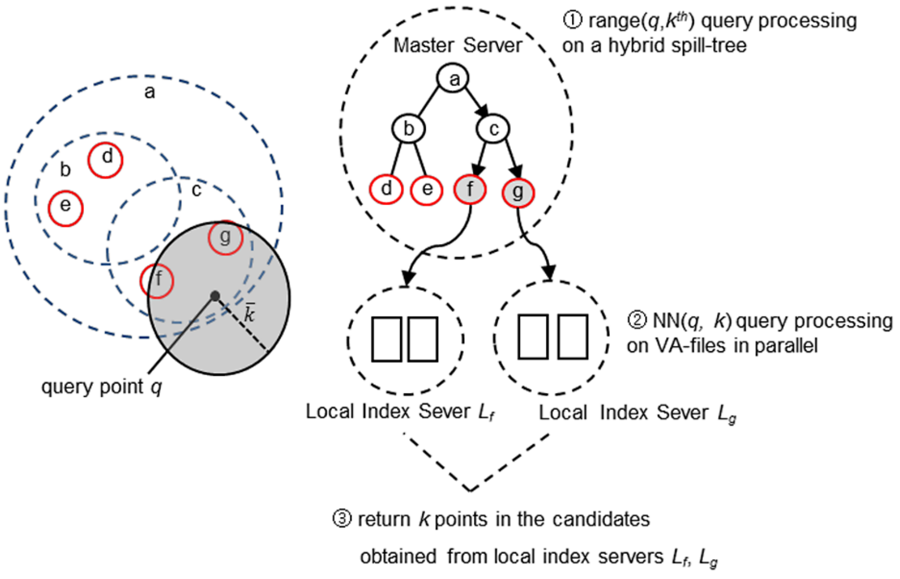

a
b
c
d
e
f
g
$\bar{k}$
query point q
Master Server
① range(q,k^{th}) query processing on a hybrid spill-tree
② NN(q, k) query processing on VA-files in parallel
Local Index Sever L_f
Local Index Sever L_g
③ return k points in the candidates obtained from local index servers L_f, L_g

Fig. 7 Processing k-NN query

Table 1 Summary of symbols and respective definitions

Symbol	Descriptions
D	Number of dimensions
Q	Query point
k	Number of nearest neighbors
$\bar{k}$	Average kth distance between points in a sample
$F(x)$	Distance distribution
P_r	Routing point stored in an internal node on a hybrid spill-tree
$r(N_r)$	Covering radius of node N_r
l	Number of nodes in a hybrid spill-tree
m	Number of leaf nodes accessed for processing a range query on a hybrid spill-tree
v	Number of points stored in a local index server
b	Number of bits used for bit-encoding (compressing) of a point
t_{vector}	Time to compute the distance between two points per dimension
t_{approx}	Time to compute lower and upper bounds per dimension on the distance between a point and a query point on a VA-file
w	Number of points remained after the filtering step of a VA-file search
t_{read}	Time to load a block from a disk
t_{compare}	Time to compare two distances

between the sample data. The master server then compares the query region with the regions of routing nodes, and returns the server routing nodes f, g with regions that intersect the query region. The k-NN query is forwarded to the data nodes L_f, L_g which are mapped to the server routing nodes f, g respectively. At each data node, a local index server returns the k nearest neighbors by an NN search based on a VA-file. The final k nearest neighbors are determined among the $2 \cdot k$ candidates that are obtained from the local index servers L_f, L_g. For instance, there are at least 300 candidates for a NN query in case when $k = 100$ and $n = 3$. Finally, we return the 100 points with the nearest distances from the query point among the 300 candidates.

The cost of k-NN query processing T_{QP} on a DVA-tree consists of the following components:

- cost for traversing a hybrid spill-tree T_{1st},
- cost for processing a k-NN query on a VA-file T_{2nd},
- cost for determining final k nearest neighbors T_{3rd}.

Relevant symbols and their descriptions are described in Table 1.

For simplicity, we assume the data points are uniformly and independently distributed in the data space. In addition, we do not consider the network transfer costs among the servers on the DVA-tree, as it is a trivial factor for measuring the total cost for k-NN query processing.

First, consider a range query range$(Q, \bar{k})$ on a hybrid spill-tree. On the hybrid spill-tree, a leaf node stores data points in corresponding region (balls) of the metric space and the related local server information. An internal (non-leaf) node stores entries with format $[P_r, r(N_r), \text{ptr}(N_r)]$, where P_r is a routing point, $r(N_r) > 0$ is a covering radius, and $\mathrm{ptr}(N_r)$ is a pointer of child node N_r. All the point P_i reachable from node N_r are in the ball of radius $r(N_r)$ centered in P_r. A node N_r of the hybrid spill-tree has to be accessed if the ball of radius $\bar{k}$ centered in the query object Q and the region associated with N_r intersect. This is the case if $d(Q, P_r) \le r(N_r) + \bar{k}$. If the distribution of distance is defined as $F(x) = \Pr\{d(P_1, P_2) \le x\}$, the probability that N_r to be accessed can be expressed as

$$\begin{aligned}\Pr\{\text{node } N_r \text{ is accessed}\} &= \Pr\bigl\{d(Q, P_r) \ge r(N_r) + \bar{k}\bigr\}\\ &= F_Q\bigl(r(N_r) + \bar{k}\bigr) \approx F\bigl(r(N_r) + \bar{k}\bigr) \qquad (6)\end{aligned}$$

The number of nodes accessed for a range query on the tree is estimated by the sum of probabilities (6) shown above over all the l nodes of the tree. That is

$$\mathrm{nodes}\bigl(\mathrm{range}(Q, \bar{k})\bigr) = \sum_{i=1}^{l} F\bigl(r(N_{r_i}) + \bar{k}\bigr) \qquad (7)$$

While searching the tree, no I/O operations are necessary, because we assume that the tree fits entirely into the main memory of the master server. Therefore, the average cost for a range query is expressed as the sum of the costs of distance computation among the query point and routing points in the accessed nodes:

$$T_{\mathrm{1st}} = \mathrm{nodes}\bigl(\mathrm{range}(Q, \bar{k})\bigr) \cdot (D \cdot t_{\mathrm{vector}}) \qquad (8)$$

The local index servers corresponding to the m leaf nodes, which are determined by the range search on the tree, process the parallel k-NN queries using the VA-files. The data points v stored in a local index server are encoded by unique bit-strings. The compressed representations of the points load into a main memory. Therefore, the cost of the first filtering step of k-NN search based on the VA-file is

$$T_{\mathrm{filtering}} = v \cdot D \cdot t_{\mathrm{approx}} \qquad (9)$$

After the filtering step, a small set of candidates remain. In the refinement step, the remaining points become a target for accessing a vector file. At this time, the disk I/O occurs by random accesses to the vector file. The number of the remaining points w is represented in [25]. The cost of the second refining step can be derived as

$$T_{\mathrm{refining}} = w \cdot (t_{\mathrm{read}} + D \cdot t_{\mathrm{vector}}) \qquad (10)$$

Finally, the total cost of the k-NN search based on the VA-file in a local index server is the sum of the costs of the two steps:

$$T_{\mathrm{2nd}} = T_{\mathrm{filtering}} + T_{\mathrm{refining}} \qquad (11)$$

After the k-NN search on the VA-file, each local index server returns a list of the k candidate points with distances to the query point. The list orders the points according to increasing distance from the query point. The $m \cdot k$ candidate points obtained from the m local index servers are merged. The merging process is to compare the distances of the $m \cdot k$ candidates and to determine final k neighbors. In the merging process, all distances must not be compared. If the kth nearest distance is found, the merging process stops. Therefore, the cost for determining final k neighbors is estimated by

$$T_{\mathrm{3rd}} = k \cdot (m - 1) \cdot t_{\mathrm{compare}} \qquad (12)$$

In conclusion, the total cost for the k-NN query processing is

$$T_{QP} = T_{1\text{st}} + T_{2\text{nd}} + T_{3\text{rd}} \tag{13}$$

With the DVA-tree, some of the critical factors that influence the k-NN search performance are a range search without backtracking on the hybrid spill-tree and a parallel k-NN search using the VA-file of the multiple local index servers.

4.3 Updating the DVA-tree

To insert a new data point, the master server initially searches the hybrid spill-tree in order to determine which VA-file(s) must be updated, and then simply forwards the new point to the selected data nodes (local index servers). Each local index server loads the approximation of the new point received from the master server into the main memory, and inserts the new point into a vector file. At this time, the new point is appended to the end of the approximation list and the vector file. If the local index server does not have free space in the main memory, both the vector data on the local index server and the corresponding leaf node of the hybrid spill-tree must be split.

For example, assume that overflow occurs in the local index server L_a related to node a as shown in Fig. 8(a). To split the leaf a of the hybrid spill-tree, we create two new child leafs b, c for the current leaf a. In this case, we apply the node splitting algorithm of a hybrid spill-tree, but perform the split based on new sample data that are randomly extracted from the vector data on the overflowed local index server L_a. Two new leafs b, c created by this split are mapped to new local index servers L_b, L_c, respectively. Note that the nodes a, b and c are server routing nodes which point toward data nodes with local index servers. After that, a new routing node a' is created to replace the server routing node a; two new server routing nodes b, c are linked as children of the new routing node a'; the sub regions covered by the server routing nodes b, c are assigned to two new data nodes L_b, L_c. In order to split the

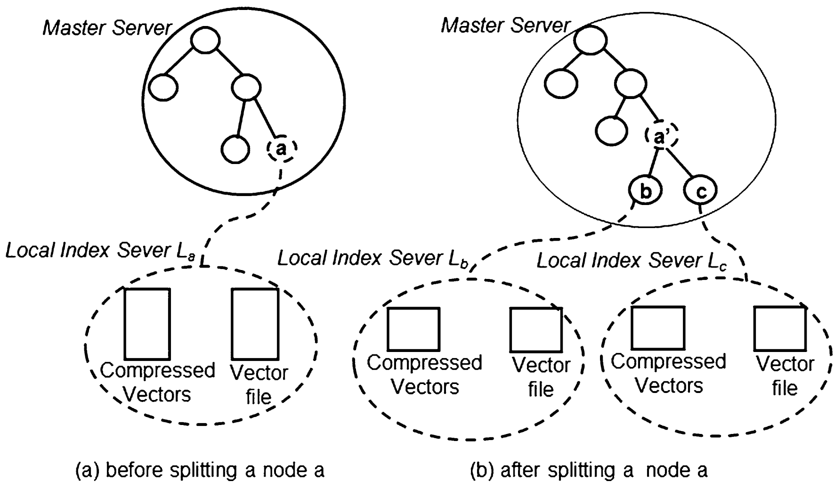


Fig. 8 Splitting a node of the DVA-tree

vector data, we sequentially scan the vector data owned by the local index server L_a. While doing so, we determine the new leaf node(s) in the updated tree where each vector belongs to. Each vector is forwarded to the new local index server(s) related to the selected leaf node(s), so the vector data on the local index server L_a can be divided into sub-regions owned by the new local index servers L_b, L_c.

At the local index server, all vectors in both a vector file and an approximation list are sequentially stored in the order which they are inserted. Whenever the compression version of a vector is removed from the approximation list, the vector located within an arbitrary position p_0 in the vector file is also deleted by a random access operation. Additionally, we restore the following vector from the location $p_0 + 1$ to the location p_0 in the vector file. Such data movement process recursively continues until we reach the end of the vector file. To avoid this problem, we perform lazy deletion. The deletion algorithm of the DVA-tree is very simple: find the local index server(s) with a vector to delete using a point query, and mark the approximation of the vector as deleted in the corresponding local index servers. The real deletion of the marked vectors can be performed by a local garbage collector which performs local restructuring of the VA-file to collect empty space in the main memory.

For each local index server, we have an update counter variable which is increased as each insert or delete operation occurs in the local index server. A local garbage collection is executed when the value of the update counter reaches a user defined threshold which may be defined as a fixed number (e.g. 500 operations). In addition, the local garbage collection can be performed if an insert operation causes an overflow of the main memory in the local index server.

While doing the local garbage collection, we do not consider a case when two leaf nodes with a parent node in the hybrid spill-tree are merged, because merge operations can cause a hybrid spill-tree restructuring. The restructuring may need to recalculate the regions of original internal nodes in the hybrid spill-tree, and to redistribute all vectors stored in multiple data nodes. Such tree restructuring is a cost expensive process, so we propose to use the local garbage collection without the merge operations between local index servers.

We note that our algorithm does not change the original internal nodes of the hybrid spill-tree, while inserting additional points or deleting points. For that reason, the hybrid spill-tree could not reflect the change of clusters by data addition and deletion. In order to keep the good query performance based on the DVA-tree, we have to rebuild the hybrid spill-tree on the sample data sets that include the original sample data and new sample data created by a node split. Basically, we can rebuild the tree once the number of update operations exceeds a pre-determined threshold, say 50 % of the original data size.

5 Experimental evaluation

In this section, we present an experimental study to evaluate the performance of the DVA-tree. The performance is evaluated using the average execution time and accuracy of a k-NN search over 100 different queries. We compare the performance of the

DVA-tree with those of the distributed hybrid spill-tree [3] and parallel cell-based filtering (CBF) [5], because they are the more recent indexing structures for the parallel NN-search in a cluster environment.

The indexing algorithms for an experimental study were developed using the M-tree C++ package [26]. We report our experimental results based on real and synthetic datasets. We use two real data sets: Corel_UCI [27] and Aerial40 [28]. The Corel_UCI contains 66,619 feature vectors with 65 dimensions of color images from the COREL library. The Aerial40 contains 270,000 feature vectors of 61 dimensions representing texture information of the large aerial photographs.

All the experiments were conducted on eight server machines that are configured with a Linux cluster, based on a global file system. Each of the eight servers had a 3.40-GHz Pentium® D CPU processor with 2.4 GB of memory capacity. For the distributed hybrid spill-tree or DVA-tree, we dedicated a master server and six other servers as local index servers that execute k-NN queries either on the local hybrid spill-tree or VA-file. Meanwhile, we used the last server as a merger to integrate the k-NN search results from the local index servers. This is to construct a similar query execution environment as the MapReduce operations for the nearest neighbor search proposed in [3]. In order to emulate a larger configuration including more than six local index servers, we also operated multiple local index servers on a single machine. The intercommunication between the master server and local index servers was done via TCP/IP.

The DVA-tree and the distributed hybrid spill-tree have top trees built upon the sample vectors. The height of the top tree is determined by the number of the sample vectors and the page capacity of leafs in the tree, which is directly related to the search delay. Therefore, for a fair performance comparison, the top trees of the DVA-tree and distributed hybrid spill-tree were built on the sample data of a same size, and their leaf pages had same capacity. Also, we used the same number of local index servers for all the indexing structures, since it is an important factor for determining parallelization of the indexing structures. The number of bits per dimension for vector compression used was eight. For the DVA-tree and distributed hybrid spill-tree, the page capacity size of leaf nodes in a top-tree was 512 KBytes.

In many applications, data points are often correlated in different ways. We first test the performance of the indexing structures on the Aerial40, which is a skewed dataset that contains 270,000 feature vectors with 61 dimensions.

Figure 9 depicts the performance of the approximate k-NN searches, as the number of the required nearest neighbors increases. The results show that the average execution time of the approximate k-NN searches on the DVA-tree runs up to 1.3 times faster than on the distributed hybrid spill-tree. Moreover, we can notice that the performance gap between the DVA-tree and the distributed hybrid spill-tree steadily widens, as the number of the nearest neighbors increases. This is due to the fact that the DVA-tree executes a top tree search using a range query processing without backtracking and a parallel VA-file search which is superior to tree-based search. Overall, both the DVA-tree and distributed hybrid spill-tree yield better performances than the parallel CBF, particularly when the number of the required nearest neighbors increases. The consequence is due to the elimination of index servers accessed. Each index server for the parallel CBF generates exactly k candidates. For instance, there

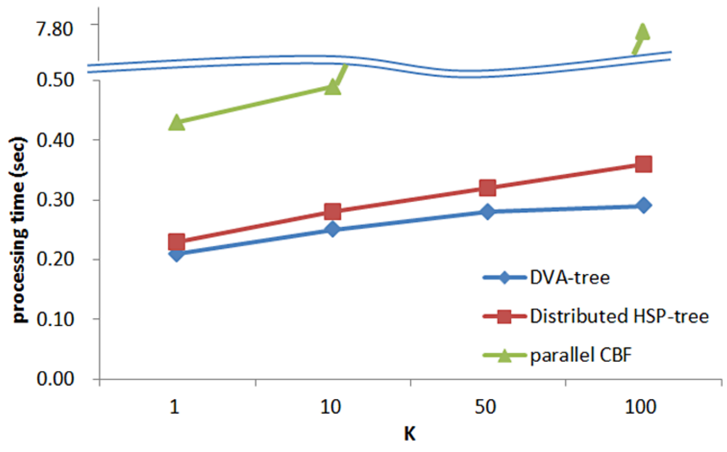


Fig. 9 The k-NN search processing delay on the skewed dataset

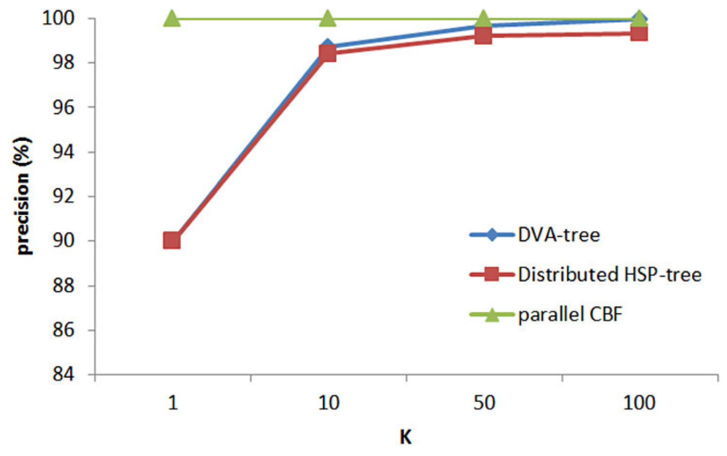


Fig. 10 The k-NN search accuracy on the skewed dataset

are at least $k \cdot n = 1{,}000$ candidates for final query results in the case of local index servers $n = 10$ and required neighbors $k = 100$. This large number of candidates considerably increases the cost of the merging step in the search of the parallel CBF.

Figure 10 shows the accuracy of the k-NN search on the skewed distributed dataset over the Aerial40. The parallel CBF aims at finding the exact NN, while the DVA-tree and distributed spill-tree only find an approximate NN. The DVA-tree obtains a better search accuracy compared to the distributed hybrid spill-tree, because it performs k-NN queries based on the VA-files in the local index servers. The accuracies of the k-NN search on the DVA-tree increase with growing number of k-neighbors required, reaching values of about 99.5 % for $k = 50$. This result is due to the fact that the DVA-tree needs to access more index servers with the increasing number of k-neighbors.

We evaluated the indexing structures using the real-data Corel_UCI of the uniformly distributed data points with 65 dimensions and the size of 66,619 points.

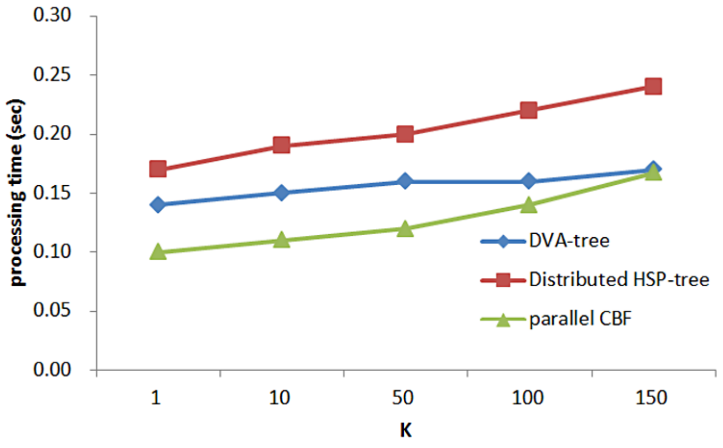


Fig. 11 The k-NN search processing delay on the uniform dataset

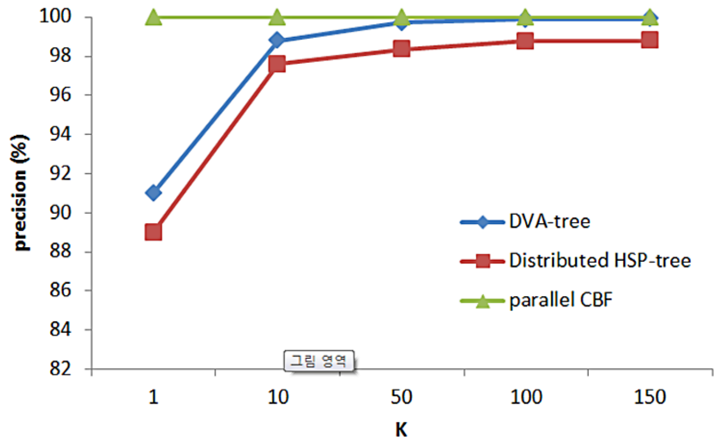


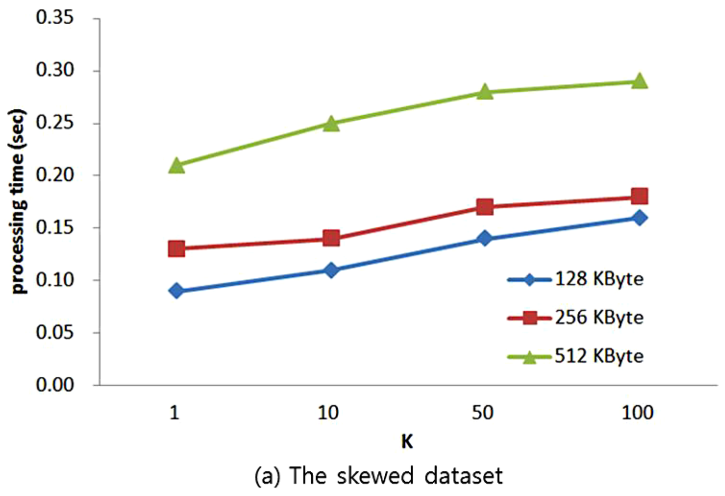
Fig. 12 The k-NN search accuracy on the uniform dataset

Figure 11 shows the results of the k-NN search on the uniformly distributed dataset. The DVA-tree runs up to 1.4 times faster than the distributed hybrid spill-tree. From these results, the DVA-tree is generally more efficient than the distributed hybrid spill-tree in a k-NN search for many types of data distribution. The parallel CBF performs very well on the uniformly distributed dataset. The parallel VA-file outperforms the DVA-tree by 1.3 times when the number of k-neighbors is 50. However, the result also illustrates that the larger k is, the less the CBF benefits from parallel execution. The cost of the parallel CBF grows linearly with the increasing number of k-neighbors. In fact, for more neighbors required ($k > 100$), the performance of the parallel CBF and the DVA-tree is almost the same.

As shown in Fig. 12, the k-NN search accuracies of the DVA-tree are still higher than that of the distributed hybrid spill-tree and are very close to that of the parallel CBF in case of $k > 50$.

From these results, we conclude that the DVA-tree is generally efficient in a *k*-NN search for many data distributions. Especially, the DVA-tree outperforms all other methods if the numbers of data points and the nearest neighbors required are sufficiently large.

For a better understanding of the behavior of the DVA-tree, we performed additional experiments on the DVA-tree only. First, we measured the times of *k*-NN query processing on the DVA-tree with varying the page capacity size of leaf nodes in the top tree. The page capacity size is an important factor that determines the number of leaf nodes. This means that the number of local index servers, which perform *k*-NN searches on the VA-files in parallel, is determined according to the page capacity size. The test data used for the experiments are two real data sets, the Aerial40 of skewed distributed data and Corel_UCI of uniformly distributed data.



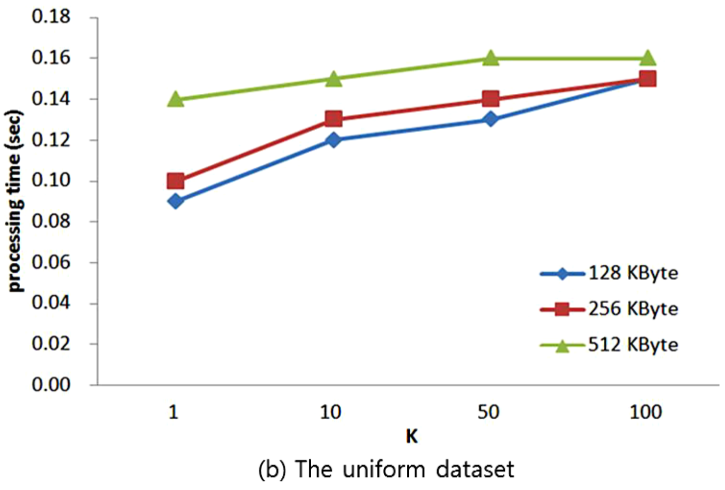


Fig. 13 The *k*-NN search processing delay of DVA-tree with varying page size of leaf nodes in the top-tree

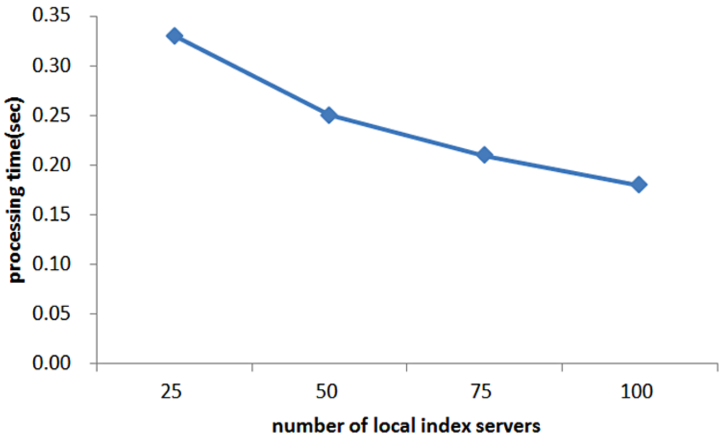


Fig. 14 The *k*-NN search processing delay varying the number of servers

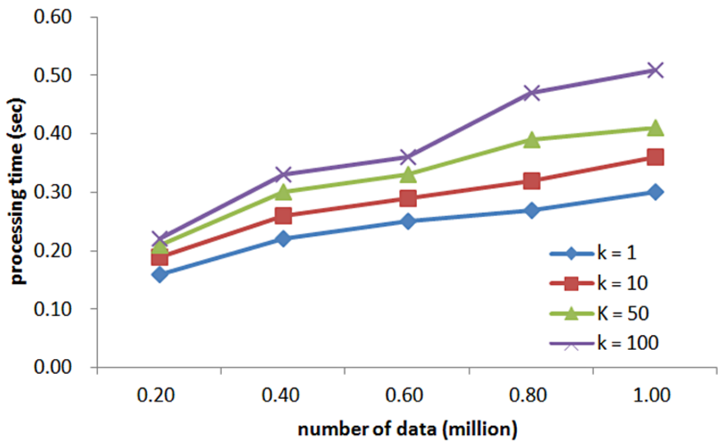


Fig. 15 The *k*-NN search processing delay for different data sizes

Figure 13 shows the average execution time of the approximate *k*-NN searches on the DVA-tree by varying the page capacity of the top tree from 128 KBytes to 512 KBytes. As shown in Fig. 13(a), the DVA-tree yields better performances on the processing time, when we use the smaller page capacity of leaf nodes in the top tree. This can be explained by the fact that the top tree with smaller page capacity of leaf nodes enables the parallel *k*-NN queries to be performed over more index servers. On the other hand, as the number of the required neighbors increases, the performance of the DVA-tree with 128 KBytes page capacity degrades more rapidly than that of the DVA-tree with 256 KBytes page capacity for uniformly distributed dataset (see Fig. 13(b)). We believe that this is because the cost of the merging process of the *k*-NN searches with the DVA-tree grows with the number of indexing servers accessed. From these experiments, we conclude that the DVA-tree is a more effective indexing structure for the skewed datasets than for the uniformly distributed datasets.

We also evaluated the scalability of the DVA-tree with respect to the number of servers. We increased the number of local index servers from 25 to 100. Since we ran this experiment on 6 machines, multiple servers ran in a single machine. For instance, with 50 servers we ran eight servers on each machine of four machines and nine servers on each machine of two machines. Because the object distribution in a high-dimensional feature space can typically be highly non-uniform, we conducted the experiment with the skewed real data set, Aerial40. As shown in Fig. 14, obviously, the cost of k-NN query processing decreases with the number of the local index servers. But the search performance does not improve linearly, because the cost of the merging process grows with the number of the local index servers. As the numbers of the local index server increased, the marginal processing delay of the k-NN search decreased.

Finally we evaluated the scalability of the DVA-tree according to the number of data. In order to measure the performance of the DVA-tree on large skewed datasets, we used the 61-dimensional synthetic data sets and increased the number of vectors up to 1,000,000. We assumed that the number of bits per dimension is 4, and the page capacity of leaf nodes in a top-tree is 512 KBytes. The number of k-neighbors varied from 1 to 100. The synthetic data were obtained by generating the desired number of points based on the Aerial40. The results are shown in Fig. 15. For 1-NN queries, the total search time was 0.16 seconds for 200,000 data and 0.30 seconds for 1,000,000 data. In case of 100-NN queries, it was 0.22 seconds for 200,000 data and 0.51 seconds for 1,000,000 data. The results of our experiments show that the DVA-tree is expected to yield a search speed increase more than linear increase of the data. Note that inaccuracy ratio was always below 5 % for all k-NN queries.

6 Conclusions

In this paper, we presented the design of a new high-dimensional indexing scheme, called a DVA-tree, to solve the distributed k-nearest neighbor search problem over large scale high-dimensional data in cluster environments. The DVA-tree employs a hierarchical clustering method and distributed VA-file management in order to allow a parallel k-NN search on each of the VA-files. We use a hybrid spill-tree as a clustering method and build the hybrid spill-tree on the sample data of large-scale high-dimensional data, because the sampling is independent of the dimensionality and the sampled data maintain the cluster information of the data set stored in the database. The data sets clustered by the hybrid spill-tree are managed on distributed VA-files. We proposed algorithms for parallel construction of a DVA-tree and approximate k-NN searches over multiple machines.

We carried out a performance evaluation of the DVA-tree on real data while comparing the DVA-tree with the distributed hybrid spill-tree and the parallel CBF. The experiments show that the DVA-tree outperforms the distributed hybrid spill-tree and parallel CBF for nearest neighbor queries on both skewed data and uniformly distributed data. Our experimental evaluations of the DVA-tree on large synthetic data indicate that the DVA-tree can efficiently support nearest neighbor queries with high accuracy. An additional advantage of the proposed indexing structure is its simplicity.

In summary, we believe that the DVA-tree is the most suitable candidate to support a high search performance, as well as index scalability for large-scale datasets. The future work includes additional implementation of the update mechanisms for the DVA-tree, and performance evaluation for larger-scale data sets.

Acknowledgements This work was supported by the IT R&D program of MKE/KEIT. [10038768, The Development of Supercomputing System for the Genome Analysis].

References

1. Nikos K, Christos F, Ibrahim K (1996) Declustering spatial databases on a multi-computer architecture. In: Proceedings of the international conference on extending database technology. LNCS, vol 1057, pp 592–614
2. Bernd S, Scott TL (1999) Master-client R-trees: a new parallel R-tree architecture. In: Proceedings of the international conference on scientific and statistical database management, pp 68–77
3. Ting L, Charles R, Henry AR (2007) Clustering billions of images with large scale nearest neighbor search. In: Proceedings of the IEEE workshop on applications of computer vision, pp 28–33
4. Roger W, Klemens B, Hans JS (2000) Interactive-time similarity search for large image collection using parallel VA-files. In: Proceedings of the European conference on research and advanced technology for digital libraries. LNCS, vol 1923, pp 83–92
5. Jaewoo C, Ahreum L (2008) Parallel high-dimensional index structure for content-based information retrieval. In: Proceedings of the IEEE international conference on computer and information technology, pp 101–106
6. Chi Z, Arvind K, Randolph YW (2004) SkipIndex: towards a scalable peer-to-peer index service for high dimensional data. Technical report TR-703-04, Princeton University
7. Beomseok N, Alan S (2005) DiST: fully decentralized indexing for querying distributed multidimensional datasets. Technical report CS-TR-4720 and UMIACS-TR-2005-28, Maryland University
8. Jagadish HV, Beng CO, Quang HV, Rong Z, Aoying Z (2006) VBI-tree: a peer-to-peer framework for supporting multi-dimensional indexing schemes. In: Proceedings of the international conference on data engineering, p 34. doi:10.1109/ICDE.2006.169
9. Mayank B, Tyson C, Prasanna G (2005) LSH forest: self-tuning indexes for similarity search. In: Proceedings of the international conference on world wide web, pp 353–366
10. Parisa H, Sebastian M, Philippe C-M, Karl A (2008) LSH at large-distributed KNN search in high dimensions. In: Proceedings of the international workshop on the web and databases
11. Roger W, Hans JS, Stephen B (1998) A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In: Proceedings of the international conference on very large data bases, pp 194–205
12. Roger W, Stephen B (1997) An approximation-based data structure for similarity search. Technical report 24, ESPRIT project HERMES (no 9141)
13. John TR (1981) The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In: Proceedings of the international ACM SIGMOD conference. doi:10.1145/582318.582321
14. David BL, Betty S (1989) A robust multi-attribute search structure. In: Proceedings of the IEEE international conference on data engineering, pp 296–304
15. Norbert B, Hans PK (1990) The R*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of the international ACM SIGMOD conference, pp 322–331
16. Stefan B, Daniel AK, Hans PK (1996) The X-tree: an index structure for high-dimensional data. In: Proceedings of the international conference on very large data bases, pp 28–39
17. Paolo C, Marco P, Pavel Z (1997) M-tree: an efficient access method for similarity search in metric spaces. In: Proceedings of the international conference on very large data bases, pp 426–435
18. Ting L, Andrew WM, Alexander G, Ke Y (2004) An investigation of practical approximate nearest neighbor algorithms. In: Proceedings of the international conference on neural information processing systems, pp 825–832
19. Christian B, Hans PK (2000) Dynamically optimizing high-dimensional index structures. In: Proceedings of the international conference on extending database technology. LNCS, vol 1777, pp 36–50
20. Guang HC, Xiaoming Z, Dragutin P, Chin WC (2002) An efficient indexing method for nearest neighbor searches in high-dimensional image databases. IEEE Trans Multimed 4(1):76–87

21. Sung GH, Jae WC (2000) A new high-dimensional index structure using a cell-based filtering technique. In: Proceedings of the international conference on database systems for advanced applications. LNCS, vol 1884, pp 79–92
22. Aristides G, Piotr I, Rajeev M (1999) Similarity search in high dimensions via hashing. In: Proceedings of the international conference on very large data bases, pp 518–529
23. Edith C, Mayur D, Shinji F, Aristides G, Piotr I, Rajeev M, Jeffrey DU, Cheng Y (2000) Finding interesting associations without support pruning. In: Proceedings of the IEEE international conference on data engineering, pp 64–78
24. Taro Y (1976) Statistics: an introductory analysis
25. Paolo C, Marco P, Pavel Z (1998) A cost model for similarity queries in metric spaces. In: Proceedings of the Australasian database conference, pp 65–76
26. http://www-deis.unibo.it/research/Mtree
27. Airphoto dataset, http://vivaldi.ece.ucsb.edu/Manjunath/research.htm
28. http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.data.html