We are IntechOpen, the world's leading publisher of Open Access books Built by scientists, for scientists

6,300

Open access books available

172,000

International authors and editors

190M

Downloads

Our authors are among the

154

Countries delivered to

TOP 1%

most cited scientists

12.2%

Contributors from top 500 universities

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index
in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

Introduction to AdaIndex—An Adaptive Similarity Search Algorithm in General Metric Spaces

Tao Ban and Youki Kadobayashi
*National Institute of Information and Communication Technology
Japan*

1. Introduction

Because of the ever-increasing amount of digital information created, captured, and replicated, we are faced today with a rapidly expanding amount of published information. It would be hard to survive this information explosion without the search engines that are readily accessible on the web, PC, and various database servers. In these systems, *similarity search* techniques (Samet, 2005; Zezula et al., 2006) always play an essential rule to efficiently retrieve user-relevant information from a large amount of data with limited computational resources. Meanwhile, similarity search has also been a prominent data preprocessing operation in fields such as data mining, signal processing, multimedia information retrieval, computational biology, pattern recognition, etc.

One popular approach for similarity search is mapping data objects into feature vectors and then conducting search in the feature space. Such a method is computationally efficient because the geometrical properties of the feature space and the vector representation of the data help to speed up the search. However, it also introduces the undesirable element of indirection into the process, especially for sophisticated applications where finding the vector representation of the data is itself an open question. A more direct approach is to define a distance function between objects and then build indexing structures based on the distance. Researches have shown that if the distance function satisfies some basic conditions, namely, non-negativity, identity, symmetry, and triangle inequality, efficient algorithms can be formulated for fast similarity search (Chávez et al., 2001; Gisli & Samet, 2003; Samet, 2005; Zezula et al., 2006). Such a distance function, usually called a metric, together with the data domain, is commonly known as a metric space. Since defining a metric between objects can be accomplished more intuitively than mapping objects to feature vectors, the metric space model has found numerous applications in processing complex data such as multimedia objects, texts, protein sequences, images, etc.

Similarity hashing methods known as Distance Index (D-index) (Dohnal et al., 2003) and its descendants incorporate multiple principles for advanced search efficiency. Assuming that the database objects are stored in the secondary storage, D-index aims to build an indexing

structure with advanced search performance by incorporating the following principles in the algorithm. First, it organizes objects into a hierarchical structure composed of multiple levels, where an individual level is further hashed into separable buckets by some split function. At query time, the levels are sequentially accessed. Because objects in difference buckets are search-separable up to some predefined margin ρ —minimum distance between objects from two separable sets—at most one bucket per level needs to be accessed for queries with search range $r \leq \rho$. Thus D-index supports a bounded search cost in the level with regards to both distance computation and disk access. Second, with the pivot filtering technique from (Micó et al., 1994; Vidal, 1994), further reduction in distance computations is achieved in the accessed bucket. According to the experiments in (Dohnal et al., 2003), D-index showed preferable results in terms of reduction of distance calculations compared with other popular metric index methods. Moreover, by storing compact clusters of objects in the separable buckets into disk pages, it also offered an excellent IO management performance.

D-index has built a good framework for metric search especially for queries with comparatively small radii. However, the pivot selection strategy suggested in (Dohnal et al., 2003) does not support balanced data partitioning in the structure and thus leads to increase in the number of distance computation and IO accesses at query time. Another problem of D-index is the tuning of parameter ρ : On the one hand, a large ρ value will lead to too many objects assigned into the inseparable regions, resulting in uncontrollable separable-set volume and number of levels. On the other hand, a small ρ value is always related with degenerated search separable property for the separable buckets. Moreover, for a complicated dataset, a ρ value uniformly performs well for all levels may even not exist.

In this paper, we present an adaptive indexing structure termed AdaIndex (Ban et al., 2009), which overcomes the above problems of D-index in the following way. When partitioning the objects into separable sets, D-index fixes the margin and leaves the formulation of the separable sets to a pivot selection procedure. On the contrary, in AdaIndex, we fix the size of the separable sets and try to maximize the margin between separable sets. The alternated strategy will lead to the following benefits. First, the number of pivots no longer influences the formulation of the separate sets so that we gain degrees of freedom to tune the number of pivots in the index structure. Thus, we can either select more pivots to improve the search performance or reduce the number to implement the index structure on a system with limited storage. Second, the maximization of the margin will result in better search performance of the pivoting rules, substantially reducing the side computations at query time. Finally, with the objects evenly partitioned into search separable sets, more efficient IO management can be supported by the algorithm.

2. Related Works on Metric Search

Before specifying the previously known search algorithms designed for metric spaces, we will first review the properties of a metric space and commonly used metric query types.

2.1 Similarity Search in Metric Spaces

Let D be the domain of objects, $d:D \times D \rightarrow \mathbb{R}$ a distance function on D . The tuple $M=(D,d)$ is called a *metric space*, if $\forall x,y,z \in D$, the following conditions hold (Brin, 1995):

$$\begin{aligned} d(x,y) &\geq 0, && \text{non-negativity;} \\ d(x,y) = 0 &\Leftrightarrow x = y, && \text{Identity;} \\ d(x,y) &= d(y,x), && \text{Symmetry;} \\ d(x,y) + d(y,z) &\geq d(x,z), && \text{triangle equality.} \end{aligned}$$

A metric query is generally defined by a query object q and a proximity condition. There are two types of queries that are widely used: the *range query* and the *nearest neighbour query*. The range query can be specified as: Given a metric space $M=(D,d)$, a finite set $X \subset D$, a query $q \in D$, and a range $r \in \mathbb{R}$, the answer set for a query q with range r is the set

$$R(q,r,X) = \{x_i | d(q,x_i) \leq r, x_i \in X\}. \quad (1)$$

Note that *point query* is a special case of range query, with $r=0$. For the nearest-neighbour query, the closet object to q is retrieved as the answer set. The concept can be easily generalized to search for the k nearest neighbours of q , thus the answer set for a k NN query is

$$K(q,k,X) = K := \{K \subseteq X, |K| = k, \forall x_i \in K, y_j \in X \setminus K, d(q,x_i) \leq d(q,y_j)\}. \quad (2)$$

For simplicity, we confine our discussion to the range query, which is the most widely explored search type. All of the discussions can be easily extended to k NN queries by maintaining a list of k candidates seen so far and using the largest distance among the candidates as the search range r in a range query. Discussions on other kinds of queries, e.g., reverse NN query, similarity join, and a combination of the enumerated search types, can be found in (Samet, 2005; Zezula et al., 2006).

2.2 Metric Search Structures

Popular indexing structures designed for similarity search in a metric space can be largely grouped into three categories: metric tree structures, distance-matrix based approaches, and hybrid methods.

The metric tree approaches include some well known methods such as the Vantage Point tree (VPT) (Yianilos, 1994), Burhard-Keller tree (BKT) (Burkhard & Keller, 1973), Generalized Hyperplane tree (GHT) (Uhlmann, 1991), Geometric Near-neighbour Access tree (GNAT) (Brin, 1995) and Metric tree (M-tree) (Ciaccia et al., 1997). When the search index is built, objects in the database are grouped into geographically adjacent clusters presented as nodes, organized into a tree-like hierarchical structure. At query time, the tree is traversed in a certain order; and to prevent direct comparison of the query object with the indexed objects, some so called pivoting rules are applied to the nodes or individual objects.

The distance-matrix based approaches refer to the methods such as AESA (Vidal, 1986; Vidal, 1994), LAESA (Micó et al., 1994), Reduced Overhead AESA (Vilar, 1995), etc., where pre-computed distances are used to characterize the indexed objects. At a preprocessing step, distances between the indexed objects and a group of selected anchoring objects, termed pivots, are computed and stored. At query time, these distances are employed to estimate unknown distances, i.e., the distances from the query object to the indexed objects, and objects which do not qualify the search criterion are skipped without direct comparison.

Distance-matrix based indexing methods provide promising performance boost in terms of distance-computation reduction. Their disadvantage lies in the enormous and sometimes infeasible space requirements. The hybrid methods try to combine both the partitioning principle, i.e., data clustering in the metric tree approaches, and the pre-computed distances technique into a single index structure. The so called D-index algorithm (Dohnal et al., 2003; Dohnal, 2004) which is an improvement on the Similarity Hashing method (SH) (Gennaro et al., 2001), is built upon a completely different principle from the metric tree approaches. D-index builds a multi-tier hashing structure, where each tier is consisted of search separable sets, organized in buckets. The structure supports easy insertion and bounded search costs for range queries up to a pre-defined search radius: at most one bucket need to be accessed at each level. Meanwhile, buckets can be arranged in to sequential disk pages so that IO cost can be saved when all objects in the bucket are all pruned during the search.

Note that there are some metric tree structures which also take advantage of the pre-computed distance, e.g., the Multi Vantage Point Tree (MVPT) (Bozkaya & Ozsoyoglu, 1997), the GNAT, and M-tree, etc. In fact, for any metric search algorithm, an extra pre-computed distance matrix can always help to substantially reduce the number of distance computations. In (Fredriksson, 2007), the author suggests applying the AESA technique to most of the existing metric search structures to efficiently reduce the number of distance computations during the search. This idea is best illustrated by the TLAESA (Gomez-Ballester et al., 2006) algorithm.

3. Metric Search by D-index

The AdaIndex algorithm introduced is partly inspired by the D-index algorithm. In this section, we will give a brief review on D-index. More detailed description of the algorithm can be found in (Dohnal et al., 2003; Dohnal, 2004).

3.1 Hashing the Dataset

In D-index, the ρ -split functions are defined to hash objects into search-separable clusters. An example is the *bps* (ball-partitioning split) function. With a predefined margin ρ , a *bps* uniquely determines the belongingness of an object $x \in X$:

$$bps^{1,\rho}(x_i) = \begin{cases} 0 & \text{if } d(x_i, p) \leq d_m - \rho, \\ 1 & \text{if } d(x_i, p) > d_m + \rho, \\ - & \text{otherwise,} \end{cases} \quad (3)$$

where p is a pivot and d_m the median of the distances from p to all $x_i \in X$. The superscript 1 denotes the order of the split function, i.e., the number of pivots involved. The subset

characterized by the symbol ‘-’ is called the *exclusion set*, noted as E . The subsets denoted by $S_{[0]}^{1,\rho}(X)$ and $S_{[1]}^{1,\rho}(X)$ are called *separable sets* according to the *separable property*:

$$d(x_i, x_j) > 2\rho, \text{ for all } x_i \in S_{[0]}^{1,\rho}(X), x_j \in S_{[1]}^{1,\rho}(X). \quad (4)$$

To partition the dataset into more separable sets, higher order ρ -split functions are composed by combining multiple first order ρ -split functions. Given m *bps* split functions, the joint m -order split function is denoted as $bps^{m,\rho}$, and the return value can be seen as a string $b=[b_1, b_2, \dots, b_m]$, where $b_i \in \{0, 1, -\}$. The following hashing operator $\langle \cdot \rangle$ returns an integer value in the range $[0, \dots, 2^m]$ for any string.

$$\langle b \rangle = \begin{cases} 2^m, & \text{if } \exists j \ b_j = -, \\ [b_1, b_2, \dots, b_m]_2 = \sum_{j=1}^m 2^{m-j} b_j & \text{otherwise.} \end{cases} \quad (5)$$

Thus through the ρ -split function and the hashing function, a mapping is defined from $x_i \in X$ to an integer $c \in [0, 2^m]$. The objects are grouped into 2^m separable subsets where the separable property still holds, and an *exclusion set*, E_H , which holds the remaining objects.

The *bps* function is defined by multiple pivots and the associated median distances. D-index applies incremental selection to select pivots. At the beginning, a set $P=\{p_1\}$ with a maximized $\mu_i (i=1)$ is chosen from the objects, where μ_i is the expectation of the inter-object distances in the feature space defined by the pivots, formally,

$$\mu_i = E_{x \in X, y \in X} \max_{s=1}^{i-1} |d(x, p_s) - d(y, p_s)|. \quad (6)$$

At step i , with the previously selected pivot set fixed, p_i is chosen from the dataset with the maximum μ_i . This process is repeated until a desired number of pivots are determined.

3.2 Storage Architecture

The storage architecture of D-index consists of a two dimensional array of *buckets* storing data objects. On the first level, an m_1 -order *bps* function is applied to the dataset and a list of separable sets are obtained. Each separable set is organized as a bucket which represents a metric region and manages all the objects falling into the region. In the next level, another *bps* function is applied to the exclusion set of the former level to form new separable sets. The algorithm repeats this procedure for a given number of times, H , or until the exclusion set cannot be further scattered. Finally, the exclusion set on the final level forms the exclusion bucket of the multi-level storage structure. To deal with overflow problems and file growth, a bucket is implemented as an elastic structure consisting of a necessary number of fixed-size *blocks*.

3.3 Search Operations

Given a D-index structure, the search algorithm guides the search procedure. For brevity, we only discuss the range search algorithm with $r \leq \rho$. Refer to (Dohnal et al., 2003) for fully specified general range search and nearest neighbour search algorithms. For a range query $R(q, r, X)$ with $r \leq \rho$, with ρ set to zero, if $bps_h^{mh,0}(q)$ yields a value smaller than 2^{mh} , only one separable bucket will be accessed at each level. Otherwise, the query sphere drops into the exclusion set and there will be some chance to skip all the separable sets in the level. Consequently, at most one separable bucket is accessed at each level. In the simple range search algorithm, we assume all levels are accessed as well as the global exclusion bucket. This algorithm requires $H+1$ bucket accesses, which is the upper bound to the more sophisticated algorithm specified in (Dohnal et al., 2003).

In D-index, special techniques are applied to speed up the search within a bucket. Generally, a bucket structure consists of a header plus a dynamic list of fixed-size blocks accommodating the objects. In the header, information on the pivots as well as the distances from all the objects in the bucket to these pivots is stored. Thus, the following pivoting rule (Vidal, 1994) can be applied to avoid unnecessary distance computations in the bucket. Let p_i be a pivot and x be an object in the bucket. Then for query $R(q, r, X)$, we have

$$|d(x, p) - d(q, p)| > r \Rightarrow d(q, x) > r. \quad (7)$$

This pivoting rule follows directly from the triangle inequality. Note that when the objects in the bucket are all pruned by the pivoting rule, bucket access can be avoided.

4. Metric Search by AdaIndex

The proposed AdaIndex is designed for similarity search in generic metric spaces. Despite of a similar hierarchical storage structure for the indexed objects, AdaIndex differentiates itself from D-index by the design philosophy. In the following, we mainly follow the terminology from D-index to save the introduction of another group of terms with similar meanings.

4.1 Partitioning the Dataset

In AdaIndex, we use the split functions to hash objects into search-separable clusters. A $Kbps$ (K -nearest-neighbour ball-partitioning split) function is defined by a tuple (p_m, K) , where p_m is a pivot and K the size of the neighbouring set to the pivot. Let the distance from p_m to its K th nearest neighbour be d_K . Then a bps uniquely determines the belongingness of any object x_i in the indexed dataset X :

$$Kbps^1(x_i) = \begin{cases} 1 & \text{if } d(x_i, p_m) \leq d_K, \\ 0 & \text{if } d(x_i, p_m) > d_K. \end{cases} \quad (8)$$

The superscript 1 denotes the order of the split function. To partition the dataset into more subsets, we compose higher order split functions by combining multiple first order $Kbps$

functions. Given M $Kbps$ functions, the joint M -order split function is denoted as $Kbps^M$, and the return value can be seen as a string $b=[b_1, b_2, \dots, b_m, \dots, b_M]$, $b_m \in \{0, 1\}$. The following $\langle \cdot \rangle$ operator returns an integer value in $[0, M]$ for any string b :

$$\langle b \rangle = \begin{cases} m, & \text{if } \exists m \text{ } b_m = 1, \\ 0, & \text{otherwise.} \end{cases} \quad (9)$$

We call the neighbouring subset associated with a pivot the separable set and the objects which do not belong to any of the separable sets are assigned to an exclusion set. If the pivots are selected to be far from each other, then we can achieve a better separable ability for the separable sets. Fig. 1a shows an example where a collection of objects are divided into three separable sets, S_1 , S_2 , S_3 , and an exclusion set E .

To make sure no object will be assigned to multiple subsets, we endow an order among the pivots—the neighbour of the previously selected pivots are removed before a subsequent pivot takes effect.

Fig. 1. AdaIndex data structure. (a) Single level partition on a 2D dataset, $K=3$, $M=3$. (b) The multilevel storage structure of AdaIndex.

4.2 Search Separable ability

In Fig. 1a, we call the minimal distance between two separable sets an adaptive margin, noted as $\rho_{i,j}$. The feature that two separable sets satisfies

$$d(x_1, x_2) > \rho_{i,j}, \quad \forall x_1 \in S_i, x_2 \in S_j. \quad (10)$$

is the so called search *separable property* in D-index. This property indicates that if the search radius $r \leq \rho_{i,j}$, then for an arbitrary query object q , only one of S_i and S_j needs to be

accessed. For example, for query q in Fig. 1a, the query sphere intersects S_2 , and because of the search separable property, objects in S_1 and S_3 can be safely pruned. Of course, there is some chance that the query sphere falls exclusively into the exclusion set and thus no separable set needs to be accessed in the level, e.g., query q' in Fig. 1a. Obviously, the margins among separable sets give a good measure on the search difficulty in the level and need to be maximized for better search performance.

4.3 Multi-level Structure

Naturally, the margins of the separable sets diminish as more pivots are added. A simple but effective approach to obtaining large margins between separable sets is to formulate a hierarchical structure. Following D-index, the data structure associated with the separable sets defined by an M -order $Kbps$ function is called a level. Note that because the objects in the exclusion set do not satisfy the search separable property so that group pruning is not supported. Moreover, as the objects in the exclusion set can not be organized into geometrically compact groups, the random access of the objects will degenerate the IO performance of the algorithm. Hence when the exclusion set of a level is still too large, we create an additional level structure with new split functions applied to form more separable sets. This process is repeated until the exclusion set is empty. Though the last level usually contains a flexible number of separable sets, in the following, we will not make difference of it from other preceding levels.

4.4 Pivot Selection

Another approach to increasing the margin of the separable sets is to incorporate some specific strategy in the pivot selection procedure. As well known is the literature, selection of pivots greatly affects the search performance for any search algorithm. We adopt the so-called farthest point clustering (Gonzalez, 1985) to select the set of M pivots, P . The pivots can be found following the algorithm specified in Alg. 1. It starts from the initialization of the pivot set with a pivot randomly selected from X (line 2); and then selects the next pivot as the object which is farthest away from the current pivot set (line 13). This process is repeated until a predefined number of pivots are chosen. When a pivot is selected, a separable set is created and the pivot together with its K nearest neighbours (line 8) is associated with the set.

```

Alg. 1 Pivot selection and separable set creation.
0 Inputs:  $X, M, K$ ;
1  $P \leftarrow \phi$ ;  $S_m \leftarrow \phi, m = 1, \dots, M$ ;
2 Randomly select  $p_1 \in X$ ;
3 for  $m = 1$  to  $M$  do
4    $P \leftarrow P \cup p_m$ ;  $X \leftarrow X \setminus p_m$ ;  $S_m \leftarrow S_m \cup p_m$ ;
5    $d_K \leftarrow d(p_m, x_K)$ , where  $x_K$  is the  $K$ th neighbor of  $p_m$  in  $X$ ;
6   for  $x_i \in X$  do
7     if  $d(x_i, p_m) \leq d_K$  then
8        $S_m \leftarrow S_m \cup x_i$ ;  $X \leftarrow X \setminus x_i$ ; // assign  $K$ NNs to the set
9     else
10       $d_i \leftarrow \min_{p_m \in P} d(x_i, p_m)$ ; // update distance to the pivot set
11    end if
12  end for
13   $p_m \leftarrow x_j$ , where  $d(x_j, P) = \max_{x_i \in X} d(x_i, P)$ ; // select next pivot
14 end for
15 return  $P, S_m$ ;

```

Alg. 1. Algorithm for pivot selection and separable set creation.

4.5 Storage Architecture

Like in D-index, AdaIndex consists of a two dimensional array of buckets which deal with IO and search operations of objects in the separable sets, as shown in Fig. 1b. In the vertical picture, the bucket lists for all H levels are ordered sequentially. In the horizontal picture at a given level, a list of separable sets is obtained from the M -order K bps function on the exclusion set of the previous level. Each of these separable sets is organized as a bucket. Based on the nature of the data, when necessary, a bucket is implemented as an elastic structure consisting of a necessary number of fixed-size blocks (disk pages), which is the basic disk access unit of the database system.

4.6 Search Operation

Given an AdaIndex structure, the range search algorithm guides the algorithm to traverse the structure for qualified objects. As shown in Alg. 2 the search algorithm sequentially traverses the H levels in the AdaIndex structure. When visiting a level, the algorithm first computes the distances from the query object to the pivots in the level and then applies the following pivoting rules to all separable sets (Fukunaga & Narendra, 1975). Let the maximal and minimal distances between a pivot p and the objects in separable set S be δ_{max} and δ_{min} , and suppose they are pre-stored in the index structure. Then, we have

Rule 1 (Fig. 2a): S contains no answer to the query, if $d(q, p) - r > \delta_{max}$;

Rule 2 (Fig. 2b): S contains no answer to the query, if $d(q, p) + r < \delta_{min}$.

If the above two rules fail at a separable set, the set is retrieved from the disk for further examination. To avoid unnecessary distance computations in the set, we make use of another two pivoting rules (Kamgar-Parsi & Kanal, 1985). Suppose the distances between an object x in the separable set and a pivot p are pre-stored in the index structure, we have

Rule 3 (Fig. 2c): x is not an answer to the query, if $d(x,p)+r < d(q,p)$;

Rule 4 (Fig. 2d): x is not an answer to the query, if $d(x,p)-r > d(q,p)$.

If Rules 3 and 4 fail at an object x , then distance $d(x,q)$ is directly computed. And when the distance function is invoked, the answer set is updated if x satisfies the query. Note that rules 3 and 4 are equivalent to the pivoting rule in equation (7).

Fig. 2. Pivoting rules. (a) Rule 1. (b) Rule 2. (c) Rule 3. (d) Rule 4.

Rules 1 and 2 are able to prune multiple objects at a time and thus we call them *group-prune rules*. Rules 3 and 4 are called *object-prune rules* since they prune one object at a time. Obviously, the object-prune rules are the special cases of group-prune rules when there is only one object contained in the separable set. If an object can be pruned by the group-prune rules, then it can be pruned by the object-prune rules as well, given the necessarily pre-stored distances. Thus application of the object-prune rules is the key to the reduction of invoked distance computations at query time. On the other hand, the group-prune rules can substantially avoid unnecessary side computations, i.e., application of the object-prune rules. Moreover, storing δ_{max} and δ_{min} is usually much cheaper than holding an array of distances in the main memory. Hence the group-prune rules also contribute to the search performance, especially for applications with computationally less intensive distance functions and tighter storage limit.

```

Alg. 2 Range search algorithm for AdaIndex.
0 Inputs:  $X, q, r$ ;
1  $A \leftarrow \emptyset$ ; // answer set
2 for  $h = 1$  to  $H$  do // sequentially traverse all levels
3    $C \leftarrow \{S_m, m = 1, \dots, M\}$ ; // yet not pruned separable sets
4   for  $m = 1$  to  $M$  do
5      $d_m = \text{metric}(q, p_m)$ ; // distance computation
6     if  $d_m \leq r$  then  $A \leftarrow A \cup p_m$ ; end if // update answer set
7     for  $l = 1$  to  $M$  do // prune separable sets by  $p_m$ 
8       if  $d_m - r > \delta_{\max}[l, m]$  or  $d_m + r < \delta_{\min}[l, m]$  then // Rules 1&2
9          $C \leftarrow C \setminus S_l$ ; // separable set pruned
10      end if
11    end for
12  end for
13  for all  $S_m \in C$  do // explore the remaining separable sets
14    Load  $x_l (l = 1, \dots, K)$  from disk; // IO access
15    for  $m = 1$  to  $M$  do // prune individual objects by  $p_m$ 
16      for  $l = 1$  to  $K$  do
17        if  $d[l, m] + r < d_m$  or  $d[l, m] - r > d_m$  then // Rules 3&4
18          else
19             $d'_l = \text{metric}(x_l, q)$ ; // distance computation
20            if  $d'_l \leq r$  then  $A \leftarrow A \cup p_m$ ; end if // update answer set
21          end if
22        end for
23      end for
24    end for
25  end for
26 return  $A$ ;

```

Alg. 2. Range search algorithm.

5. Experiments

In this section, we will evaluate the efficacy of the AdaIndex algorithm. The performance of the index structures is measured by three criteria, i.e., distance calculations, disk accesses, and side computations. In each of the reported experiments, the indexed set, the validation set, and the query set are independently drawn from the same uniform distribution in N -dimensional unit hyper-cubes. The coordinates of the data points are never directly used and only the inter-object distances are taken as inputs to the algorithm. We set the indexed dataset size to 10,000, the validation set size to 100, and the query set size to 1,000. If not otherwise specified, the search range is set to retrieve about 5% of the data from the database and the results are averaged over 100 queries for the validation and 1,000 queries for testing.

Fig. 3. Parameter selection of M and K based on different criteria. (a) Distance computation. (b) Disk access. (c) Side computation. (d) Disk computation plus side computation. (e) Integrated overall cost. (f) Selected parameters for 2-10 dimensional simulation datasets.

5.1 Parameter Selection

There are three parameters in AdaIndex, i.e., the number of pivots in the level, M , the size of a separable set, K , and the number of levels in the structure, H . Because H is uniquely determined by M and K , hereafter we only discuss how to tune M and K . To find appropriate values of M and K with better search efficiency, we conduct a grid search process: Given a list of values for M and a list for K , for each possible pair of parameters, we build an AdaIndex structure and test its performance on a validation set. The parameters with the best search performance are employed to build an index structure for evaluation on the test set.

In the following, we illustrate the above parameter tuning process of a 10-D dataset. The distance computations, disk accesses, and side computations against M and K averaged over the validation set are shown as contours in Fig. 3. In Fig. 3a, we can see that M is the deterministic factor for the number of distance computations at query time. The reason is obvious: M determines the storage cost of the index structure, and intuitively, more stored information in the data structure will lead to more prevented distance computations. Fig. 3b shows the number of disk access for difference parameter settings. Here, we assume an ideal situation that a separable set can always be stored in a disk page. And thus it is obvious that the larger the value of K , the less disk accesses during the search. However, in a practical application where the object size is not ignorable to the size of a disk page, the contours may vary a lot. Consider the other extreme case that an object always takes one or more disk pages. Obviously, this time the number of disk accesses will correlate exactly to the number of distance computations. From the above two extreme cases, we can see that for an application, whatever the ratio of the object size to the disk-page size, the actual contours of disk accesses will feature some combined properties in Fig. 3a and Fig. 3b. In fact, the larger the ratio of the object size to disk-page size, the flatter the slope of the contours. Fig. 3c presents the contour of side computations during the search. Similar as in Fig. 3a, the number of pivots is the deterministic factor for side computations. However, at this time, the effect of adding more pivots are minus—the more pivots employed the more side computations invoked during the search.

Above we have shown some basic principals for parameter tuning for a given evaluation criterion. It is worth noting that in a real application, computational cost of the search usually involves more than one factor from the above. However, the selection of parameters based on multiple criteria seems somewhat contradictory. Thus it will be interesting to explore some strategy on parameter tuning on multiple criteria. We adopt the weighting technique to solve this problem. For a given application, it is easy to estimate a pair of weight coefficients α and β such that the overall computation cost of a query can be computed as

$$C = \alpha C_{dc} + \beta C_{da} + C_{sc}. \quad (11)$$

Here, C_{sc} is the number of side computations which are taken as reference, C_{dc} the number of distance computations, and C_{da} the number of disk accesses. Base on the above definition, we can again employ the grid search method to select a group of parameters which gives optimal computational cost for the application. Fig. 3d shows the contours of the overall computational cost for $\alpha=1,000$ and $\beta=0$, i.e., the IO costs are neglected. We can

see that for a quite wide range of parameters, i.e., $M \in [80, 500]$ and $K \in [40, 450]$, the search performance is near optimal. Fig. 3e shows the result when the IO costs are considered by setting $\beta = 10,000$. The acceptable range of parameter shrinks to be around $M \in [110, 370]$ and $K \in [250, 450]$.

Fig. 4. Search performance measured by overall costs. (a) On a 5-D dataset with dynamic search range. (b) On a 5-D datasets with varying sample size. (c) On multi-dimensional datasets.

5.2 Performance Evaluation

In this section, we evaluate the search performance of AdaIndex on uniformly distributed data in N -dimensional Euclidean spaces. Fig. 3f shows the selected parameters for $N = 2, \dots, 10$. We set $\alpha = 1,000$ and $\beta = 10,000$ for all datasets. And for reference, we also report the results of D-index.

In the first experiment, we show how the search range r influences the search performance of AdaIndex. The evaluation is done on a 5-D dataset. The search range is gradually increased with the answer set growing from about 0.2% to 20% of the indexed set. As shown in Fig. 4a, the overall cost of AdaIndex increases almost linearly with the search range,

especially when the search range is small. For all the search ranges, AdaIndex results in much less overall cost than D-index since it avoids more distance computations during the search.

Then we test the performance of AdaIndex for datasets with varying sample size. AdaIndex and D-index are built on 5-D datasets up to size 20,000. In the evaluation, we fix the search range for retrieving about 5% of the data. The curves of the overall cost against the dataset size are shown in Fig. 4b. It is obvious that the number of distance calculations for AdaIndex increases linearly as indexed data in the structure increases. Compared with D-index, the search performance of AdaIndex is more preferable.

The aim of the last experiment is to show the performance of AdaIndex on datasets with different dimensions. The algorithms are tested against datasets with dimensions up to 10. It is easy to learn from Fig. 4c that the search difficulty increases gradually as the dimension grows because of the so called curse of dimensionality. In spite of the increased difficulty, AdaIndex again shows a much better performance than D-index.

Above all, AdaIndex has shown stable performance in a variety of simulated experiments. For all the tested datasets, it always has a preferable performance than D-index.

6. Conclusion

In this chapter, we have presented the AdaIndex, an adaptive index algorithm for efficient similarity search in general metric spaces. AdaIndex has incorporated the following principles to speed up the search. First, it borrows the idea from D-index to create a hierarchical data structure so that the search can be efficiently conducted in parallel mode. Second, on an individual level, the objects are partitioned into compact clusters and are managed by the bucket structure for effective IO management. Then, with the maximized search separable property among the buckets and the advanced pivoting rules, AdaIndex supports significant reduction in distance computations. Another very important property of the AdaIndex algorithm is its adaptivity for varying applications. Given a group of cost coefficients for different types of computational cost, we can easily apply the introduced grid search method for an index structure with optimal overall performance.

7. References

- Ban, T.; Guo, S.; Xu, Q.; & Kadobayashi, Y. (2009). AdaIndex: an adaptive index structure for fastsimilarity search in metric spaces, *Lecture Notes in Computer Science*, Vol. 5864, pp. 729–737.
- Bozkaya, T. & Ozsoyoglu, Z. M. (1997). Distance-based indexing for high-dimensional metric spaces, *Proceedings of the ACM International Conference on Management of Data (SIGMOD 1997)*, pp. 357–368, ACM Press.
- Brin, S. (1995). Near neighbor search in largemetric spaces, *Proceedings of the 21st International Conference on Very Large Data Bases*, pp. 574–584.
- Burkhard, W. A. & Keller, R. M. (1973). Some approaches to best-match file searching, *Communications of the ACM*, Vol. 16, No. 4, pp. 230–236.

- Chávez, E.; Navarro, G.; & Marroquin, J. L. (2001). Searching in Metric Spaces, *ACM Computing Surveys*, Vol. 33, No. 3, pp. 273–321.
- Ciaccia, P.; Patella, M.; & Zezula, P. (1997). M-tree: an efficient access method for similarity search in metric spaces, *Proceedings of the 23rd International Conference on Very Large Data Bases*, pp. 426–435.
- Dohnal, V.; Gennaro, C.; Savino, P.; & Zezula, P. (2003). D-Index: distance searching index for metric data sets, *Multimedia Tools and Applications*, Vol. 21, No. 1, pp. 9–33.
- Dohnal, V. (2004). *Indexing Structures for Searching in Metric Spaces*, PhD thesis, Faculty of Informatics, Masaryk University in Brno, Czech Republic.
- Fredriksson, K. (2007). Engineering efficient metric indexes, *Pattern Recognition Letters*, Vol. 28, No. 1, pp. 75–84.
- Fukunaga, L. & Narendra, P. M. (1975). A branch and bound algorithm for computing k-nearest neighbors, *IEEE Transactions on Computers*, Vol. 24, No. 7, pp. 750–753.
- Gennaro, C.; Savino, P.; & Zezula, P. (2001). Similarity search in metric databases through hashing, *Proceedings of the 3rd ACM Multimedia 2001 Workshop on Multimedia Information Retrieval (MIR 2001)*, pp. 1–5.
- Gisli, H. R. & Samet, H. (2003). Index-driven similarity search in metric spaces, *ACM Transactions on Database Systems*, Vol. 28, pp. 517–580.
- Gomez-Ballester E.; Micó L.; & Oncina J. (2006). Some approaches to improve tree-based nearest neighbor search algorithms, *Pattern Recognition Letters*, Vol. 39, pp. 171–179.
- Gonzalez, T. F. (1985). Clustering to minimize the maximum intercluster distance, *Theoretical Computer Science*, Vol. 38, pp. 293–306.
- Kamgar-Parsi, B. & Kanal, L. N. (1985). An improved branch and bound algorithm for computing k-nearest neighbors, *Pattern Recognition Letters*, Vol. 3, No. 1, pp. 7–12.
- Micó, M. L.; Oncina, J.; & Vidal, E. (1994). A new version of the nearest-neighbor approximating and eliminating search algorithm (AESAs) with linear preprocessing time and memory requirements, *Pattern Recognition Letters*, Vol. 15, No. 1, pp. 9–17.
- Samet, H. (2005). *Foundations of Multidimensional and Metric Data Structures*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- Uhlmann, J. K. (1991). Satisfying general proximity/similarity queries with metric trees, *Information Processing Letters*, Vol. 40, No. 4, pp. 175–179.
- Vidal, E. (1986). An algorithm for finding nearest neighbors in (approximately) constant average time, *Pattern Recognition Letters*, Vol. 4, pp. 145–157.
- Vidal, E. (1994). New formulation and improvements of the nearest-neighbor approximating and eliminating search algorithm (AESAs), *Pattern Recognition Letters*, Vol. 15, No. 1, pp. 1–7.
- Vilar, J. M. (1995). Reducing the overhead of the AESA metric space nearest neighbor searching algorithm, *Information Processing Letters*, Vol. 56, No. 5, pp. 265–271.
- Yianilos, P. N. (1993). Data structures and algorithms for nearest neighbor search in general metric spaces, *ACM-SIAM Symposium on Discrete Algorithms 1993*, pp. 311–321.
- Zezula, P.; Amato, G.; Dohnal, V.; & Batko, M. (2006). *Similarity Search – The Metric Space Approach*, Springer.

Advances in Computer Science and IT

Edited by D M Akbar Hussain

ISBN 978-953-7619-51-0

Hard cover, 420 pages

Publisher InTech

Published online 01, December, 2009

Published in print edition December, 2009

The book presents some very interesting and excellent articles for this divergent title. The 22 chapters presented here cover core topics of computer science such as visualization of large databases, security, ontology, user interface, graphs, object oriented software developments, and on the engineering side filtering, motion dynamics, adaptive fuzzy logic, and hyper static mechanical systems. It also covers topics which are combination of computer science and engineering such as meta computing, future mobiles, colour image analysis, relative representation and recognition, and neural networks. The book will serve a unique purpose through these multi-disciplined topics to share different but interesting views on each of these topics.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Tao Ban and Youki Kadobayashi (2009). Introduction to AdaIndex — an Adaptive Similarity Search Algorithm in General Metric Spaces, *Advances in Computer Science and IT*, D M Akbar Hussain (Ed.), ISBN: 978-953-7619-51-0, InTech, Available from: <http://www.intechopen.com/books/advances-in-computer-science-and-it/introduction-to-adaindex-an-adaptive-similarity-search-algorithm-in-general-metric-spaces>

INTECH
open science | open minds

InTech Europe

University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China

Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

© 2009 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the [Creative Commons Attribution-NonCommercial-ShareAlike-3.0 License](https://creativecommons.org/licenses/by-nc-sa/3.0/), which permits use, distribution and reproduction for non-commercial purposes, provided the original is properly cited and derivative works building on this content are distributed under the same license.

IntechOpen

IntechOpen