An Access Structure for Similarity Search in Metric Spaces

Vlastislav Dohnal

Faculty of Informatics, Masaryk University,
Botanická 68a, 602 00 Brno, Czech Republic
xdohnal@fi.muni.cz

Abstract. Similarity retrieval is an important paradigm for searching in environments where exact match has little meaning. Moreover, in order to enlarge the set of data types for which the similarity search can efficiently be performed, the mathematical notion of metric space provides a useful abstraction of similarity. In this paper, we present a novel access structure for similarity search in arbitrary metric spaces, called D-Index. D-Index supports easy insertions and deletions and bounded search costs for range queries with radius up to ρ. D-Index also supports disk memories, thus, it is able to deal with large archives. However, the partitioning principles employed in the D-Index are not very optimal since they produce high number of empty partitions. We propose several strategies of partitioning and, finally, compare them.

1 Introduction

Searching has always been one of the most prominent data processing operations because of its useful purpose of delivering required information efficiently. However, exact match retrieval, typical for traditional databases, is not sufficient or feasible for present applications, such as multimedia information retrieval, data mining, machine learning, and genome databases. What seems to be more useful, if not necessary, is to base the search paradigm on a form of *proximity*, or *dissimilarity* of a query and data objects. Roughly speaking, objects that are near a given query object form the query response set. In this place, the notion of mathematical *metric space* provides a useful abstraction of nearness.

Several storage structures, such as [2, 3, 6, 9], have been designed to support efficient similarity search execution over large collections of metric data where only a distance measure of pairs of objects is possible to quantify. However, the performance is still not satisfactory. Though all of the indexes are trees, many tree branches have to be traversed to solve a query, because similarity queries are basically range queries that typically cover data from several tree nodes or some other content-specific partitions. In principle, if many data partitions need to be accessed for a query region, they are not *contrasted* enough and such partitioning is not useful from the search point of view. Recently, the *excluded middle vantage point* strategy for partitioning of metric data has been proposed to develop an index structure [12] called *Excluded Middle Vantage Point Forest*, which creates a forest of one path search trees.

In this article, we describe recently proposed similarity search structure, called D-Index [8], which uses this strategy. The organization stores data objects in buckets with direct access to avoid hierarchical bucket dependencies. Such organization also results in a very efficient insertion and deletion of objects. Though similarity constraints of queries can be defined arbitrarily, the structure is extremely efficient for queries searching for very close objects. We point out issues related to the design of the D-Index. In particular, we propose three strategies to combine partitioning principles which lead to better behavior of D-Index.

2 Related Work

The urgent need of indexing techniques that support execution of similarity queries led to the application of *spatial access methods* (SAMs) such as R-Tree. Unsatisfactory performance of R-Trees on high dimensional vector spaces lead to the further development. New access structures for searching high dimensional spaces, e.g. iDistance [13] or Pyramid [1] were proposed and analyzed. They both outperform R-Trees and demonstrate significant speedup. However, SAMs are limited by the following assumptions on which they rely: i) objects are represented by vectors in a multidimensional vector space, ii) the similarity between a pair of objects is often based on an L_p metric, e.g. Euclidean distance, which does not introduce any kind of correlation. Moreover, SAMs assume that the distance function can be trivially evaluated by means of time, which is not always the case of multimedia data, e.g. the distance of strings is measured by the *edit distance*, which has the complexity $\mathcal{O}(n^2)$. Finally, SAMs usually optimize only the number of access to disk memories, not the number of distance evaluations.

A more general approach to the similarity based searching is an index structure that operates in metric spaces. Note that in metric spaces we can only state the distance between two objects and a metric function is often considered as a CPU demanding operation. Uhlmann [10] proposes *metric tree* that partitions the metric space using the relative distances of objects. This technique is improved in [11] and *vantage point tree* is proposed. A further improvement of this concept has recently been presented by Yianilos [12]. This approach is based on the *excluded middle vantage point strategy* and creates a forest of vantage point trees. This principle is also exploited in the D-Index. The main contribution of indexing methods based on metric spaces is that they are not limited to the usage on vector spaces, thus, they can be applied on other domains, such as text strings or XML documents, without any additional nontrivial transformation to a vector space.

Although the number of access structures for metric spaces is impressive, see a recent survey [5] most of them suffer from being intrinsically *static*, which limits their applicability in dynamic environments. Contrary to SAM, presented metric trees optimize only the number of distance computations. Zezula et al proposed M-Tree [6] that optimizes both CPU and I/O costs.

3 Searching in Metric Spaces

A convenient way to assess similarity between two objects is to apply metric functions to decide the closeness of objects as a distance, that is the objects' dissimilarity. A *metric space* $\mathcal{M} = (\mathcal{D}, d)$ is defined by a domain of objects (elements, points) $\mathcal{D}$

and a total (distance) function d – a *non-negative* ($d(x,y) \geq 0$ with $d(x,y) = 0$ iff $x = y$) and *symmetric* ($d(x,y) = d(y,x)$) function that satisfies the *triangle inequality* ($d(x,y) \leq d(x,z) + d(z,y)$, $\forall x,y,z \in \mathcal{D}$). We assume that the maximum distance never exceeds d^+, thus we consider a *bounded metric space*.

In general, the problem of indexing in metric spaces can be defined as follows: *given a set $X \subseteq \mathcal{D}$ in the metric space $\mathcal{M}$, preprocess or structure the elements of X so that similarity queries can be answered efficiently*. For a query object $q \in \mathcal{D}$, two fundamental similarity queries can be defined. A *range query* retrieves all elements within distance r to q, that is, the set $\{x \in X, d(q,x) \leq r\}$. A *nearest neighbor* query retrieves the h closest elements to q, that is a set $R \subseteq X$ such that $|R| = h$ and $\forall x \in R, y \in X - R, d(q,x) \leq d(q,y)$.

For the space constraints of this article, we consider only the similarity range search operations here. In the following, we first define partitioning principles and then outline the ideas of the D-index. Next, we provide the range search algorithms. Finally, we present a sketch of comparison with M-Tree.

3.1 General Approach: Separable Partitioning

To achieve the objectives, the partitioning principle of D-Index is based on a mapping function, which is called the *ρ-split function*, where ρ is a real number constrained as $0 \leq \rho < d^+$. In order to gradually explain the concept of ρ-split functions, we first define a first order ρ-split function and its properties.

Definition 1. *Given a metric space $(\mathcal{D}, d)$, a first order ρ-split function $s^{1,\rho}$ is the mapping $s^{1,\rho} : \mathcal{D} \to \{0, 1, -\}$, such that for arbitrary different objects $x, y \in \mathcal{D}$, $s^{1,\rho}(x) = 0 \ \wedge \ s^{1,\rho}(y) = 1 \ \Rightarrow \ d(x,y) > 2\rho$ (separable property) and $\rho_2 \geq \rho_1 \ \wedge s^{1,\rho_2}(x) \neq - \ \wedge \ s^{1,\rho_1}(y) = - \ \Rightarrow d(x,y) > \rho_2 - \rho_1$ (symmetry property).*

In other words, the ρ-split function assigns to each object of the space $\mathcal{D}$ one of the symbols 0, 1, or $-$.

The concept of ρ-split functions can, of course, be generalized by concatenating n first order ρ-split functions with the purpose of obtaining a split function of order n.

Definition 2. *Given n first order ρ-split functions $s_1^{1,\rho}, \ldots, s_n^{1,\rho}$ in the metric space $(\mathcal{D}, d)$, a ρ-split function of order n $s^{n,\rho} = (s_1^{1,\rho}, s_2^{1,\rho}, \ldots, s_n^{1,\rho}) : \mathcal{D} \to \{0, 1, -\}^n$ is the mapping, such that for arbitrary different objects $x, y \in \mathcal{D}$, $\forall i \ s_i^{1,\rho}(x) \neq - \ \wedge \ s_i^{1,\rho}(y) \neq - \ \wedge \ s_i^{n,\rho}(x) \neq s_i^{n,\rho}(y) \ \Rightarrow \ d(x,y) > 2\rho$ (separable property) and $\rho_2 \geq \rho_1 \ \wedge \forall i \ s_i^{1,\rho_2}(x) \neq - \ \wedge \ \exists j \ s_j^{1,\rho_1}(y) = - \ \Rightarrow d(x,y) > \rho_2 - \rho_1$ (symmetry property).*

An obvious consequence of the ρ-split function definitions is that by combining n first order ρ-split functions $s_1^{1,\rho}, \ldots, s_n^{1,\rho}$, which satisfy the separable and symmetric properties, we obtain a ρ-split function of order n $s^{n,\rho}$ which also demonstrates the separable and symmetric properties. We often refer to the number of symbols generated by $s^{n,\rho}$, that is the parameter n, as the *order* of the ρ-split function. In order to obtain an addressing scheme with direct access, another function that transforms the ρ-split strings into integers is defined as follows.

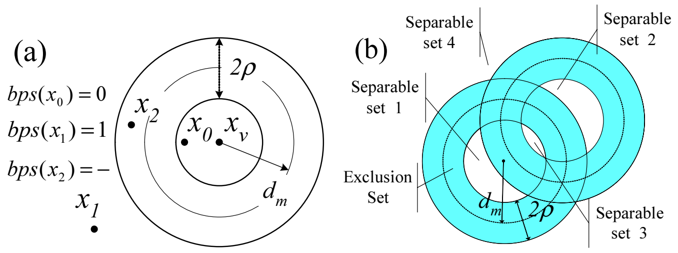

Fig. 1. The *bps* split function (a) and the combination of two *bps* functions (b)

Definition 3. *Given a string* $b = (b_1, \ldots, b_n)$ *of* n *elements 0, 1, or* $-$*, the function* $\langle\cdot\rangle : \{0, 1, -\}^n \rightarrow [0..2^n]$ *is specified as:*

$$\langle b\rangle = \begin{cases} [b_1, b_2, \ldots, b_n]_2 = \sum_{j=1}^{n} 2^{n-j} b_j, \textit{ if } \forall j\ b_j \neq - \\ 2^n, \textit{ otherwise} \end{cases}$$

When all the elements are different from '$-$', the function $\langle b\rangle$ simply translates the string b into an integer by interpreting it as a binary number (which is always $< 2^n$), otherwise the function returns 2^n.

By means of the ρ-split function and the $\langle\cdot\rangle$ operator, we can assign an integer number i ($0 \leq i \leq 2^n$) to each object $x \in \mathcal{D}$, i.e., the function can group objects from $X \subset \mathcal{D}$ in $2^n + 1$ disjoint subsets.

Though several different types of first order ρ-split functions are proposed, analyzed, and evaluated in [7], the *ball partitioning split* (*bps*) originally proposed in [12] under the name *excluded middle partitioning*, provided the smallest exclusion set. For this reason, this approach is also applied in the D-Index and can be characterized as follows.

The ball partitioning ρ-split function *bps* uses one reference object (pivot) x_v and the *medium distance* d_m to partition a data set into three subsets, see Figure 1a. The result of the *bps* function gives the unique identification of the set to which the object x belongs:

$$bps(x) = \begin{cases} 0 & \text{if } d(x, x_v) \leq d_m - \rho \\ 1 & \text{if } d(x, x_v) > d_m + \rho \\ - & \text{otherwise} \end{cases}$$

The subset of objects characterized by the symbol '$-$' is called the *exclusion set*, while the subsets of objects characterized by the symbols 0 and 1 are the *separable sets*, because any range query with radius not larger than ρ cannot find qualifying objects in both the subsets.

To obtain more separable sets we define higher order ρ-split function as a combination of several *bps* functions, where the resulting exclusion set is the union of the exclusion sets of the original split functions. Furthermore, the new separable sets are obtained as the intersection of all possible pairs of separable sets of the original functions. Figure 1b gives an illustration of this idea for the case of two split functions. Several strategies of combining ρ-split functions are discussed in Section 4.

3.2 D-Index

The basic idea of the D-Index is to create a multilevel storage and retrieval structure that uses several ρ-split functions, one for each level, to create an array of *buckets* for storing objects. On the first level, we use a ρ-split function for separating objects of the whole data set. For any other level, objects mapped to the exclusion bucket of the previous level are the candidates for storage in separable buckets of this level. Finally, the exclusion bucket of the last level forms the exclusion bucket of the whole D-Index structure. It is worth noting that the ρ-split functions of individual levels use the same ρ. Moreover, split functions can have different order, typically decreasing with the level, allowing the D-Index structure to have levels with a different number of buckets. In particular, from the structure point of view you can observe the buckets organized as the following two dimensional array consisting of $1 + \sum_{i=1}^{h} 2^{m_i}$ elements.

$$\begin{array}{l} B_{1,0}, B_{1,1}, \ldots, B_{1,2^{m_1}-1} \\ B_{2,0}, B_{2,1}, \ldots, B_{2,2^{m_2}-1} \\ \vdots \\ B_{h,0}, B_{h,1}, \ldots, B_{h,2^{m_h}-1}, E_h \end{array}$$

All separable buckets are included, but only the E_h exclusion bucket is present – exclusion buckets $E_{i<h}$ are recursively re-partitioned on level $i+1$. Then, for each row i (i.e. the D-Index level), 2^{m_i} buckets are defined and are separable up to 2ρ, thus we are sure that there do not exist two buckets at the same level i both containing relevant objects for any similarity range query with radius $r \leq \rho$.

In order to deal with overflow problems and growing files, buckets are implemented as *elastic buckets* and consist of the necessary number of fixed-size blocks (pages) – basic disk access units.

Range Search. Given a range query $\mathcal{Q} = \mathcal{R}(q, r)$, we define a simple search algorithm. This algorithm, however, evaluates only limited queries with $r \leq \rho$.

Algorithm 1. *Search*

```
for $i = 1$ to $h$
    return all objects $x$ such that $x \in \mathcal{Q} \cap B_{i,\langle s_i^{m_i,0}(q)\rangle}$;
end for
return all objects $x$ such that $x \in \mathcal{Q} \cap E_h$;
```

During the elaboration of the algorithm we manipulate the value of parameter ρ of split functions. When we use $\rho = 0$ the function $\langle s_i^{m_i,0}(q)\rangle$ always gives a value smaller than 2^{m_i}. Consequently, one separable bucket on each level i is determined. Finally, the algorithm also accesses the exclusion bucket of the whole structure. The algorithm requires $h+1$ bucket accesses, which forms the upper bound on the search.

Generic Range Search. Algorithm 1 requires to access one bucket at each level of the D-Index, plus the exclusion bucket. In the following two situations, however, the number of accesses can even be reduced: i) if the query region is contained in the exclusion partition of the level i, then the query cannot have objects in the separable buckets of this level and only the next level must be considered, ii) if the query region is contained in a separable partition of the level i the following levels, as well as, the exclusion bucket need not be accessed, thus the search terminates on this level.

Another drawback of the simple algorithm is that it works only for search radii up to ρ. However, with additional computational effort queries with $r > \rho$ can also be executed. Indeed, such queries can be executed by evaluating the split function $s^{r_q-\rho}$. In case $s^{r_q-\rho}$ returns a string without any '$-$', the result is contained in the single bucket $B_{\langle s^{r_q-\rho}\rangle}$ plus, possibly, the exclusion bucket.

Let us now consider that the string returned contains at least one '$-$'. We indicate this string as $b = (b_1, \ldots, b_n)$ with $b_i = \{0, 1, -\}$. In case there is only one $b_i =$ '$-$', we must access all buckets B, whose index is obtained by substituting'$-$' with 0 and 1. For this purpose, we define the function G that returns all identifications of buckets that must be accessed. For example, G returns the set $\{\langle 001\rangle, \langle 011\rangle\}$ for the string '0–1'. In the most general case, we must substitute in the string b all the '$-$' with zeros and ones and generate all possible combinations.

Given a query region $\mathcal{Q} = \mathcal{R}(q, r_q)$ with $q \in \mathcal{D}$ and $r_q \leq d^+$. An advanced algorithm can execute the similarity range query as follows.

Algorithm 2. *Range Search*

```
01. for i=1 to h
02.     if $\langle s_i^{m_i,\rho+r_q}(q)\rangle < 2^{m_i}$ then (exclusively in a separable bucket)
03.         return all objects $x$ such that $x \in \mathcal{Q} \cap B_{i,\langle s_i^{m_i,\rho+r_q}(q)\rangle}$; exit;
04.     end if
05.     if $r_q \leq \rho$ then (search radius up to $\rho$)
06.         if $\langle s_i^{m_i,\rho-r_q}(q)\rangle < 2^{m_i}$ then(not exclusively in an exclusion b.)
07.             return all objects $x$ such that $x \in \mathcal{Q} \cap B_{i,\langle s_i^{m_i,\rho-r_q}(q)\rangle}$;
08.         end if
09.     else (search radius greater than $\rho$)
10.         let$\{l_1, l_2, \ldots, l_k\} = G(s_i^{m_i,r_q-\rho}(q))$
11.         return all objects $x$ such that $x \in \mathcal{Q} \cap B_{i,l_1}, \ldots, x \in \mathcal{Q} \cap B_{i,l_k}$;
12.     end if
13. end for
14. return all objects $x$ such that $x \in \mathcal{Q} \cap E_h$;
```

In general, Algorithm 2 considers all D-Index levels and eventually also accesses the global exclusion bucket. However, the test on the line 02 can discover the exclusive containment of the query region in a separable bucket and terminate the search earlier. Otherwise, the algorithm proceeds according to the size of the query radius. If $r \leq \rho$ there are two possibilities. If the test on line 06 is satisfied one separable bucket is accessed. Otherwise no separable bucket is accessed on this level because the query region is from this level point of view exclusively in the exclusion zone. Provided the search radius is greater than ρ, more separable buckets are accessed on a specific level. Unless terminated earlier, the algorithm accesses the exclusion bucket at line 14.

3.3 Comparison

We have compared the D-Index with other index structures, particularly, we considered M-Tree[1] [6] and a sequential organization (SEQ). According to [5], these are the only types of index structures for metric data that use disk memories to store objects. We have conducted the experiments on 45-dimensional vectors of image color features compared by the *quadratic distance* measure. The data set consisted of 11,000 objects and had practically normal distribution. We have measured average performance over 50 different query objects considering numerous similarity range queries. The results are shown in Figure 2.

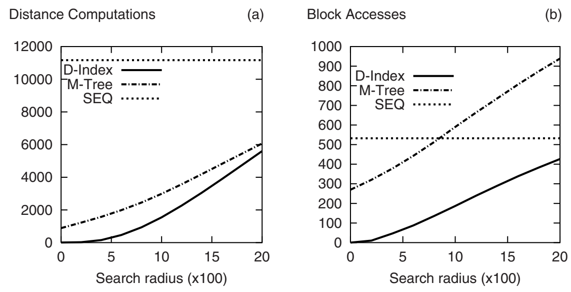

Fig. 2. Comparison of the range search efficiency in the number of distance computations (a) and the number of block accesses (b)

For all tested queries, i.e. retrieving subsets up to 20% of the database, the D-Index always needed less distance computations than the M-tree and the number of block accesses of the M-tree was significantly higher than for the D-Index. This is obvious since the M-Tree has nodes with fixed capacity[2] while the bucket in D-Index consists of necessary number of blocks (the size of blocks is fixed). The superior performance in the terms of distance computations can be attributed to the fact that the D-Index also applies the pivot-based filtering techniques, which significantly reduce the number of distance computation, for details see [8]. Moreover, the D-Index uses the same pivots in the partitioning, i.e. ρ-split functions, and in the filtering technique to further reduce the number of distance evaluations. Figure 2b demonstrates another interesting observation: to run the *exact match* query, i.e. range search with $r = 0$, the D-Index only needs to access one block. As a comparison, the M-Tree needs one half of the SEQ. Notice that the exact match search is used to locate an object to be deleted and forms the main cost of delete operations. In this respect, the D-Index is able to manage deletions much more efficiently than the M-tree. In this case, the advantage of the D-Index over the M-Tree is caused by its structure. The M-Tree must traverse the tree and access all internal nodes along the search path. However, the D-Index computes ρ-split functions, which does

[1] The software is available at http://www-db.deis.unibo.it/research/Mtree/

[2] Overflow problems are solved with node splits. Thus, after a split there is approximately 50% of space wasted in the new nodes, which need not be filled anymore.

not require any disk accesses and directly determines the bucket to access. Due to the applied pivot-based algorithm, all object in blocks of buckets are sorted according to the distance to a pivot. Consequently, the D-Index is able to locate a block in the bucket where the object to delete resides without any additional block accesses.

We have also performed scalability tests over the same data collection but ranging from 100,000 to 600,000 objects. For these experiments, the D-Index structure was defined by 37 pivots and 74 buckets.The D-Index was strictly linear, that is, the costs to evaluate a query grow linearly with the data size. The M-Tree was slightly better than the D-Index but this is attributed to the fact that M-Tree is incrementally reorganizing its structure while the D-Index structure is using a constant structure. The development of a dynamic structure of D-Index is our main research issue. This problem is partially discussed in the next section.

4 Improved Partitioning Strategies

In this stage, we emphasize the main drawback of the D-Index access structure and sketch some possible solutions. The experiments revealed that the D-Index is very efficient and outperforms the others nearly in all situations. However, its partitioning principles are not very optimal and produce unbalanced partitions. In particular, we concern the problem of selecting reference objects (pivots). Next, we deal with the issue of combining several rho-split functions into a single mapping function.

The problem of choosing pivots is important for any search technique in the general metric space, because all such algorithms need, directly or indirectly, some "anchors" for partitioning and search pruning. It is well known that the way in which pivots are selected affects the performance of proper algorithms. This has been recognized and demonstrated by several researchers [2, 11]. Recently, the problem was systematically studied in [4], and several strategies for selecting pivots have been proposed and tested. The generic conclusion is that good pivots are i) far away from the remaining objects of the metric space and ii) far away from each other pivot. In the D-Index, we also use this technique to select pivots.

The design of D-Index structure requires specification of several ρ-split functions which are usually combinations of *bps* functions. In general, the idealized split function should produce balanced buckets each containing nearly the same number of objects and minimize the size of the exclusion bucket. Figure 3 presents three possible strategies to combine two *bps* functions. The first technique, depicted in (a) is utilized in the D-Index and uses the pivot p_1 and $d_m = r_1$ to divide the space into two separable partitions. Next, these two partitions are repartitioned using a different *bps* function which applies a different pivot p_2 and $d_m = r_2$, however, the same for both the partitions. As a result, we obtain four separable buckets. We refer to this method as the *strict strategy*.

The second strategy in (b) differs from the first one in one aspect. It makes use of the pivot p_2 in the second function as well, but two different values of d_m, r_2^1 and r_2^2 are applied for the left and the right partition, respectively. The hypothesis behind is that by manipulations with the parameter d_m we can achieve better balanced buckets, diminish empty buckets and decrease the occupation of exclusion sets. We refer to this as the *variable d_m strategy*.

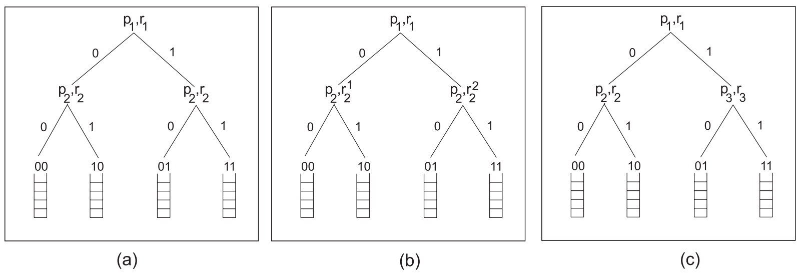

Fig. 3. Different strategies of combining two *bps* split functions

To complete the list of split policies, we introduce the third strategy which modifies also the pivot. In details, this technique can be viewed as the application of three independent *bps* functions instead of two in the previous methods. The first *bps* function specified with the pivot p_1 and radius r_1 produces two separable partitions. These two parts are separately divided using the next two *bps* functions, that is, the left one uses *bps* function with p_2 and r_2 while the right set is split using p_3 and r_3. In this way, we get four separable buckets. However, the major disadvantage of this approach is enormous memory requirements, e.g. for 128 separable buckets we need 127 different pivots, which is in a sharp contrast with the other strategies that need $\log(128) = 7$ pivots only to define the same number of buckets. The memory requirements are not the only issue: the more pivots we have the more distance computations we must evaluate during both insert and search operations. Since we optimize the costs in terms of distance computations, such the behavior is not desired. For this reason, we neglect this strategy. In the following, we compare the first two approaches, strict and variable d_m strategies.

4.1 Experimental Evaluation

In order to test properties of strict and variable d_m strategies, we have considered the collection of 45-dimensional vectors consisting of 11,000 objects again. We created two one-level D-Index structures one for each strategy and organized the dataset. At this point, it is important to remark that buckets are fixed in size, specifically, each bucket is able to store 500 objects at maximum. Table 1 shows the results for both methods. We have examined several properties: the number of needed pivots – $|P|$, the symbol $|E|$ denotes the percentage of exclusion bucket occupation, $|B|$ is the average occupation of separable buckets, the number of empty buckets is denoted by $|\{\}|$, and the number of non-empty buckets is $2^{|P|} - |\{\}|$.

Table 1. Results for different strategies of combining split functions

strategy	$\|P\|$	$\|E\|$	$\|B\|$	$\|\{\}\|$	$2^{\|P\|} - \|\{\}\|$
strict	9	36.40%	27.16%	463	49
variable d_m	5	33.76%	48.23%	0	32

The strict strategy applied 9 pivots and produced the structure consisting of 512 buckets. The average bucket occupation $|E|$ was 27.16%, that is, about 136 objects accommodated in a bucket. The remaining objects fell into the exclusion set, thus, the occupation of exclusion bucket was 36.40%. The worst parameter observed was that the strict strategy produced 463 empty buckets, which is 90.40% of all buckets. This is in a sharp contrast with the requirements for a good split function which should produce balanced split.

The variable d_m technique performed much better. The number of pivots used to partition the dataset was only 5, which leads to 32 buckets in total. The average bucket occupation was 48.23% that is much higher than that for the strict strategy. The occupation of exclusion bucket decreased to 33.76%. Finally, the most promising fact is that the variable d_m diminished all empty buckets.

All in all, the results confirm our hypothesis that the variable d_m strategy leads to a better balanced partitioning. It also reduces the occupation of the exclusion bucket, moreover, it diminishes all empty buckets. To sum up, the variable d_m strategy outperforms the strict strategy and uses much less pivots to achieve the same partitioning. On the other hand, the variable d_m strategy introduces additional computational overhead, however, these costs do not include any supplementary distance computations and can be neglected. To conclude, the variable d_m strategy is promising and we will implement it to create a dynamic structure of the D-Index.

5 Concluding Remarks

Metric spaces have recently become an important paradigm for similarity search and many index structures supporting execution of similarity queries have been proposed. However, most of the existing structures are limited to operate in main memory only, so they do not scale up to high volumes of data. We have concentrated on the case where indexed data are stored on disks and we have compared a novel index structure for similarity range and nearest neighbor queries called D-Index with other disk-based approaches. Contrary to other index structures, such as the M-tree, the D-Index stores and deletes any object with one block access cost, so it is particularly suitable for dynamic data environments. Compared to the M-tree, it typically needs less distance computations and much less disk accesses to execute a query. We have also discussed the design issues of the D-Index and proposed several partitioning strategies. The experiments revealed that the *variable d_m strategy* provides the best partitioning.

We will concentrate on possibilities of implementing a dynamic structure of the D-Index that would allow automatic updates of split functions to achieve an optimal design of structure for various applications. We will report results in the near future. The multilevel hashing structure of the D-Index inherently offers parallel and distributed implementations. This is our next research direction.

References

1. Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *ACM SIGMOD 1998*, pages 142–153, 1998.

2. Tolga Bozkaya and Z. Meral Özsoyoglu. Indexing large metric spaces for similarity search queries. *ACM TODS*, 24(3):361–404, 1999.
3. Sergey Brin. Near neighbor search in large metric spaces. In *VLDB 1995*, pages 574–584, 1995.
4. Benjamin Bustos, Gonzalo Navarro, and Edgar Chávez. Pivot selection techniques for proximity searching in metric spaces. In *SCCC 2001, Proceedings of the XXI Conference of the Chilean Computer Science Society*, pages 33–40. IEEE CS Press, 2001.
5. Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquin. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.
6. Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB 1997*, pages 426–435, 1997.
7. Vlastislav Dohnal, Claudio Gennaro, Pasquale Savino, and Pavel Zezula. Separable splits in metric data sets. In *Proceedings of 9-th Italian Symposium on Advanced Database Systems, SEBD 2001*, pages 45–62, 2001.
8. Vlastislav Dohnal, Claudio Gennaro, Pasquale Savino, and Pavel Zezula. D-Index: Distance searching index for metric data sets. *Multimedia Tools and Applications*, 21(1):9–33, 2003.
9. Caetano Traina Jr., Agma J. M. Traina, Bernhard Seeger, and Christos Faloutsos. Slim-Trees: High performance metric trees minimizing overlap between nodes. In *EDBT 2000*, volume 1777, pages 51–65, 2000.
10. Jeffrey K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, 1991.
11. Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 311–321, 1993.
12. Peter N. Yianilos. Excluded middle vantage point forests for nearest neighbor search. In *6th DIMACS Implementation Challenge, ALENEX'99*, 1999.
13. Cui Yu, Beng Chin Ooi, Kian-Lee Tan, and H. V. Jagadish. Indexing the distance: An efficient method to KNN processing. In *VLDB 2001*, pages 421–430, 2001.