



# A Parallel Implementation of GHB Tree

Zineddine Kouahla<sup>1(✉)</sup> and Adeel Anjum<sup>2(✉)</sup>

<sup>1</sup> LABSTIC, University of 08 Mai 1945 Guelma, Guelma, Algeria  
kouahlazinedine@yahoo.fr

<sup>2</sup> Department of Computer Science,  
COMSATS Institute of Information Technology, Islamabad, Pakistan  
adeel.anjum@comsats.edu.pk

**Abstract.** Searching in a dataset remains a fundamental problem for many applications. The general purpose of many similarity measures is to focus the search on as few elements as possible to find the answer. The current indexing techniques divides the target dataset into subsets. However, in large amounts of data, the volume of these regions explodes, which will affect search algorithms. The research tends to degenerate into a complete analysis of the data set. In this paper, we proposed a new indexing technique called GHB-tree. The first idea, is to limit the volume of the space. The goal is to eliminate some objects without the need to compute their relative distances to a query object. Peer-to-peer networks (P2P) are superimposed networks that connect independent computers (also known as nodes or peers). GHB-tree has been optimized for secondary memory in peer-to-peer networks. We proposed a parallel search algorithm on a set of real machine. We also discussed the effectiveness of construction and search algorithms, as well as the quality of the index.

## 1 Introduction

Efficient indexing is an increasingly important area in computer science. Indexing techniques have been improved to deal with searches on large collections of data. However, it has been found that the indexing processes become more difficult. It is difficult to compare these techniques [1–3], their effectiveness depend on different factors (type of data, quality of the computing machine, etc.). Formally, a metric space is defined for a family of elements that are comparable through a given distance. The distance function measures the dissimilarity between two elements from a given database, in such a way that smaller distances correspond to more similar elements. Let  $\mathcal{O}$  be a set of elements. Let  $d : \mathcal{O} \times \mathcal{O} \rightarrow \mathbb{R}^+$  be a distance function, which verifies: (i) non-negativity:  $\forall (x, y) \in \mathcal{O}^2, d(x, y) \geq 0$ , (ii) reflexivity:  $\forall x \in \mathcal{O}, d(x, x) = 0$ , (iii) symmetry:  $\forall (x, y) \in \mathcal{O}^2, d(x, y) = d(y, x)$ , and (iv) triangle inequality:  $\forall (x, y, z) \in \mathcal{O}^3, d(x, y) + d(y, z) \leq d(x, z)$ . The concept of metric space is rather simple and leads to a limited number of possibilities for querying an actual database of such elements. These are called similarity queries and several variants exist. We consider  $k$  nearest neighbor

( $k$ NN) searches, i.e., searching for the  $k$  closest objects with respect to a query object. There are two main types of similarity queries: the range and the  $k$ -nearest neighbor queries.

Let  $q \in \mathcal{O}$  be a query point and  $k \in \mathbb{N}$  be the expected number of answers.

Then  $(\mathcal{O}, d, q, k)$  defines a  $k$ NN query, the value of which is  $S \subseteq \mathcal{O}$  such that  $|S| = k$  (unless  $|\mathcal{O}| < k$ ) and  $\forall (s, o) \in S \times \mathcal{O}, d(q, s) \leq d(q, o)$ .

The main factor that influence the efficiency of search algorithms, when the dimension increases is called the “dimensionality-curse problem”. The current methods have proven to be unreliable, it becomes hard to store, manage, and analyze this amount of data. This problem is caused by inherent deficiencies of space partitioning, and also, the overlap factor between regions that will influence subsequent performance search algorithms. So the problem is still open. An efficient structure is based on a better grouping of similar objects in compact clusters. In a previous work [4], we led our researches on indexing via tree structure. It is based on the successive division of the space with the spheres. It is a technique that leads to simpler data structures, and therefore simple algorithms.

Moreover, on large scale, the regions of balls become very large, which could degenerate the index. This subsequently reflects on the search algorithm. Distributed (P2P) systems, which are framed by similarly advantaged hubs interfacing with each other in a self-sorting out way, have been a stand-out amongst the most critical models for information sharing. The main difficulty that one faces when searching is a generalized version of the so-called “multidimensional curse problem”. When distances tend to be close to each other, the objects become almost indistinguishable, they cannot be grouped into clearly separated clusters and, as reported by several authors, searches tend to degenerate into full scans of the whole data set. These remarks open two possible directions (that can possibly be combined as we discuss below): improve sequential scans; provide parallel algorithms.

Our proposed system is based on the use of the most efficient indexing structure in the peer-to-peer (P2P) network, which are formed by equally privileged nodes connecting to each other in a self-organizing way, and have been one of the most important architectures for data sharing. While P2P networks are well known for their efficiency, scalability and robustness.

The rest of the paper is organized as follows. Section 2 of the paper provides an overview of Bag of features and adaboost algorithm. Section 3 deals with the proposed algorithm and Sect. 4 deals with the experimental analysis and Sect. 5 concludes the paper.

## 2 Background

Based on these two partitioning techniques, the first class does not enforce a partitioning of the space. The M-tree [5] builds a balanced index, allows incremental updates. On the context of the reorganization of objects in compact clusters, Almeida [6] proposed a new structure but just for an approximate search, called Divisive-Agglomerative Hierarchical Clustering or DAHC-tree. In [7], the

authors proposed an extension of Slim-Tree named Slim\*-tree, that exploits the best properties from ball and the BST as a hash function to search within a bucket file. The problem has not been resolved and the reinsertion of objects remains costly on a large scale. A novel clustering based dynamic indexing and retrieval approach is proposed, termed as CD-Tree [3], updates the structure with constant insertion of data. The nodes in the CD-Tree are fitted by Gaussian Mixture Models. In our opinion, the problem is not totally solved because the update of construction phase remains slow, and becomes costly on a large scale. The second class is based on the partitioning of the space. There have been a number of longitudinal studies [8,9]. Two sub-approaches are included: the first uses ball partitioning, like VP-tree [10–12]. In this method, the choice of the pivots plays a very important role on the index structure, that is why, Yianilos proposes the VP-tree [10], it is based on finding the median element of a set of objects. The mVP-tree is a generalization of the VP-tree, the nodes are divided into quantiles. This principle of partitioning eliminates the problem of overlapping between shapes. However, in this type of approach, a problem arises in cases where a demand point is close to the border between two regions; it is necessary to visit all the neighboring regions which makes the index less efficient. Combine two trees to improve the search time, an idea that has been proposed by Curtin [13], it uses the kd-tree and ball-tree to take advantage of both information. Several difficulties were cited by the authors. The main problem is that the efficiency decreases if the dimension is greater than 10. Other techniques [14] have been proposed in the last two years trying to index large-scale data but does not meet the exact but approximate queries, and other try to compress the index [15]. This leaves the door open to other proposals in the future.

In our previous work [4], we led our researches on indexing via tree structure. It is based on the successive division of the space with the spheres. Moreover, with the large amount of current data, the region of balls become very large, which could degenerate the index. This subsequently reflects on the search algorithm. This problem is caused by inherent deficiencies of space partitioning, and also, the overlap factor between regions. This is one of the major problems in this type of work. The Parallelism can and should certainly be part of a solution. We believe that no technical indexing can achieve a logarithmic search time. We know that a logarithmic response time is achievable with parallel implementations. On sequential version, a general sort is  $O(n \cdot \log(n))$ , where  $n$  is the number of objects. On a parallel machine, a sort can be implemented in  $O(\log(n))$  time and  $O(n)$  on the surface, namely the number of processors.

### 3 The Proposed GHB-tree

The partitioning of space is a technique that leads to simpler data structures - hence algorithms. Moreover, the problem of exponentially increasing volumes in large spaces argues in favor of techniques that would otherwise reduce or at least limit volumes, or even control their occupancy. We introduce a new structure called GHB-tree (*Generalised Hyper-plane Bucketed*) [16], inspired from

**Algorithm 1.** Insertion in GHB-tree

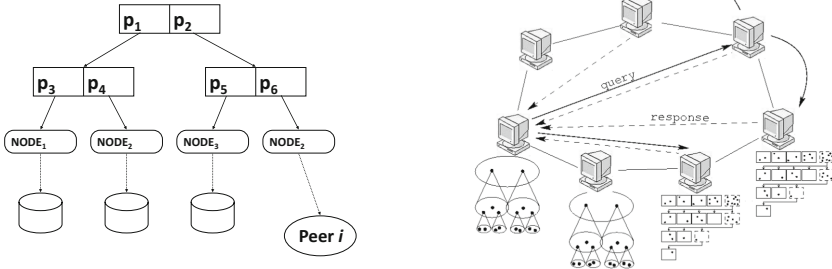
$$\text{Insert-GHB} \left( \begin{array}{l} o \in \mathcal{O}, \\ N \in \mathcal{N}, \\ c_{\max} \in \mathbb{N}^*, \end{array} \right) \in \mathcal{N}$$

$$\triangleq \begin{cases} (o, \perp, \perp, \perp) & \text{if } N = \perp \\ (p_1, o, \perp, \perp) & \text{if } N = (p_1, \perp, \perp, \perp) \\ (p_1, p_2, \text{Insert}(o, d, c_{\max}, G), D) & \text{if } N = (p_1, p_2, L, R) \wedge \\ & d(p_1, o) \leq d(p_2, o) \\ (p_1, p_2, L, \text{Insert}(o, d, c_{\max}, D)) & \text{if } N = (p_1, p_2, L, R) \wedge \\ & d(p_1, o) > d(p_2, o) \end{cases}$$

GH-tree. The Fig. 1 illustrates the development of a tree. At each stage of the recursive process of constructing the tree, two pivots are chosen from a subset of elements  $c_{\max}$ , they are chosen as the two objects furthest apart from each other. First, a node  $Nodes_{GHB}$  - or only  $\mathcal{N}$  - consists of two elements and two children:

$$(p_1, p_2, L, R) \in E \times E \times \mathcal{N}_{GHB} \times \mathcal{N}_{GHB}. \quad (1)$$

or:  $p_1, p_2$  are two non-confused elements,  $d(p_1, p_2) > 0$ , called “pivots”, they thus define a hyper-plane;  $L$  and  $R$  are the subtrees associated with the elements respectively in the “left” parts. A (sub) tree can be empty, which is denoted by  $\perp$ .



**Fig. 1.** Parallel version of GHB-tree

**Construction of a GHB-tree.** Building a GHB-tree is realised incrementally. The insertion is done in a top-down way. Algorithm 1 describes formally the incremental insertion process. When the cardinal limit is reached, a leaf is replaced by an inner node. Besides, the tree tends to be rather balanced, hence inserting a new object is a logarithmic operation, in amortised cost. This algorithm is implemented in the order of balancing network peer loads.

**Algorithm 2.** Search kNN in GHB-tree

$$\text{kNN-GHB} \left( \begin{array}{l} N \in \mathcal{N}, \\ q \in \mathbb{R}^n, \\ k \in \mathbb{N}^*, \\ d \in \mathcal{O} \times \mathcal{O} \rightarrow \mathbb{R}^+, \\ r_q \in \mathbb{R}^+ = +\infty, \\ A \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}} = \emptyset \end{array} \right) \in (\mathbb{R}^+ \times \mathcal{O})^{\mathbb{N}}$$

with :

- $A^L = \text{kNN-GHB}(L, q, k, d, r_q, k\text{-insertion}(A, ((d(p_1, q), p))))$ ;
- $A^R = \text{kNN-GHB}(R, q, k, d, r_q, A)$ ;
- $r_q^L = \max\{d : (d, o) \in A^G\}$  if  $|A^L| = k$  else  $r_q$ ;
- $r_q^R = \max\{d : (d, o) \in A^D\}$  if  $|A^R| = k$  else  $r_q$ ;
- $A^{LR} = \text{kNN-GHB}(O, q, k, d, r_q^G, A^G)$ ;
- $A^{RL}$ ;

$$\triangleq \left\{ \begin{array}{l} A \quad \text{if } N = \perp \\ A^L \quad \text{if } N = (p, r, L, R) \wedge \\ \quad d(q, p_1) - r_q^L < d(q, p_2) \wedge \\ \quad d(q, p_1) < d(q, p_2) \\ A^R \quad \text{if } N = (p, r, L, R) \wedge \\ \quad d(q, p_1) - r_q^R > d(q, p_2) \wedge \\ \quad d(q, p_1) > d(q, p_2) \\ A^{LR} \quad \text{if } N = (p, r, L, R) \wedge \\ \quad d(q, p_1) - r_q^L \leq d(q, p_2) + r_q^L \wedge \\ \quad d(q, p_1) < d(q, p_2) \\ A^{RL} \quad \text{if } N = (p, r, L, R) \wedge \\ \quad d(q, p_1) + r_q^R \geq d(q, p_2) - r_q^R \wedge \\ \quad d(q, p_1) > d(q, p_2) \end{array} \right.$$

We have considered putting in place strategies to try to balance the tree, such as choosing two elements furthest apart from each other. However, we are careful not to use a function of more than linear complexity, otherwise the algorithm will exceed a complexity in  $O(n \cdot \log n)$  which is the one it has in this version.

**kNN Search in GHB-tree.** The Algorithm 2, which formally describes the search kNN in a GHB-tree, is also quite complex. The searches are made from balls while the space has been partitioned. The search is done by calculating the distance between the query point and the two pivots, while descending into the tree. Not counting the case of the empty tree, we can meet four cases when passing through a tree node:

- The first case is where the search result is located entirely in the left subtree. In other words, the search ball lies entirely in the left half-plane. Similarly, the second case is where the search result is fully present in the right subtree.

- The third and fourth cases are those where the search must *a priori* be continued in the two subtrees because the search ball overlaps the two half-spaces. What distinguishes the third of the fourth case is the position of the center. If the center is in the left hyperplane, then the search will continue first in the left son. Only if the search has not sufficiently reduced the radius of the search ball will the pursuit take place in the right son. The search can be modified *a posteriori* to finally get back to the first case. Obviously, the fourth case is where the search in both threads is reversed.

Note that this algorithm is the same on all the stations of the network. It is on this logical network that the query  $q$  is broadcast. In each time the indexes are browsed, the value of the query radius  $r_q$  decreases, which actually corresponds to the distance to the  $k^e$  object in the ordered list  $A$ .

The leaf nodes contain a subset of the indexed data with a maximum cardinal equal to  $c_{\max}$ . At the leaf level the procedure is quite simple. In order to find the  $k$  closest neighbours of a leaf, just sort them according to their increasing distances to the  $q$  request object. Then we return *at most* the first  $k$  sorted items. Note that a real sort is not necessary; there is a variant, called  $\ll k\text{-sort} \gg$ , which is only in:  $O(c_{\max} \cdot \log_2 k)$ . Note that  $c_{\max}$  being either a constant, a logarithm of the size of the collection, or its square root, the complexity of the operation on a sheet is very fast, or even constant. The  $r_q$  query radius plays the essential role for search optimization (the minimum possible is a maximum of pruning). It is initially set to  $+\infty$  by default, but we *hopefully* see it dwindle with each move on an internal node.

Note, again, that this step does not really require sorting, but only a sequence of mergers. The complexity is “constant”, that is to say in:

$$O(2.k) \tag{2}$$

rather than:

$$O(2.k \cdot \log_2 k). \tag{3}$$

## 4 Experiments and Comparison

In this section we provide experimental results on the performance of GHB-tree on real data sets, in order to test and compare its effectiveness. We used tow datasets. We started with the cities of France, which have a low dimensional. We turned to the complex objects, good example is multimedia descriptors, we used a subset of the MPEG-7 Dominant Color Descriptor (KDD), it can be found at <http://kdd.ics.uci.edu>. We run our structure with same datasets on a workstations computers with the configuration Intel(R) Xeon(R) CPUs, and 8 GB of main memory. All index files were stored on a network partition.

We arrange the size of each tree node to be equal to the size of a disk page. We compared ourselves to the MM-tree [17] its extension onion-tree, as well as slim\* [7], an improved version of the M-tree [5], and IM-tree [4]. We used the library C++ “GBDI Arboretum” which implements these methods and we

adapt them to be executable in a P2P environment.<sup>1</sup> In Fig. 2, we see that our proposal is the most effective compared to others with a difference of over 30% with the onion-tree in the three collections (average and with the two values of  $c_{\max}$ ), and with more than 40% compared to the Slim\*-tree. The difference from MM and onion-trees is easily explainable. The reason is the absence of the respective “semi-balancing algorithm” and “keep-small” that require a number of additional operations. In slim\*-tree “slim-down algorithm” also has a significant cost which was noted by its own authors [7]. Our approach, GHB-tree is simple in the insertion of new objects, (which was one of the initial objectives with respect to the complexity  $O(n \log n)$  reasonable) and provides an incremental index competitive.

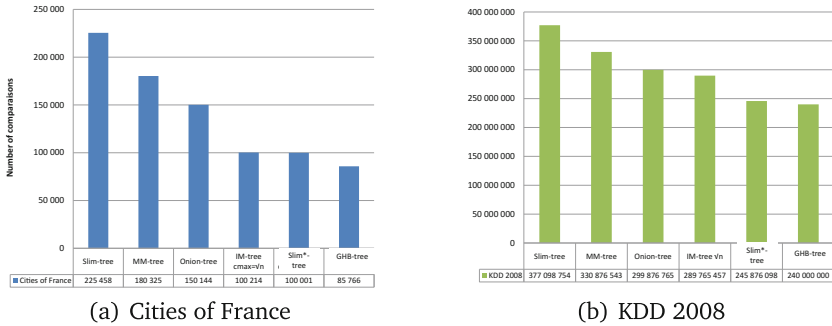


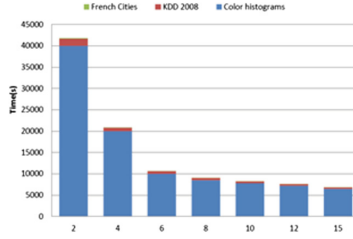
Fig. 2. Performance statistics of construction algorithms in GHB-tree

We vary in different ways the  $k$ NN searches. Firstly, for building the index, we run with different values of  $c_{\max}$  parameter which was chosen either as the square root or as the logarithm of the size of the collection. Next, we run  $k$ NN searches with  $k$  between 5 and 100. We note also that the difference between the perfect version which is the most effective, and sequential versions is not negligible. This allows us to say that there is a possibility of minimizing this gap using techniques to find as soon as possible nearest neighbor with a minimum of energy.

This proves that the creation of the index has been beneficial by the creation of dense, even at large scale and also in large amount of data. For the parameter  $k$ , we observe that if we increase its value of  $k$ , the performances decreases but with a gap between less than 1% and to less than 2% when  $k = 50$ . So the value of  $k$  has no major influence on the performance of the search algorithm. Figure 3 shows the elapsed time for building indexes for each of the three collections.

As shown from the figure, the elapsed time gradually decreases as the number of cores increases from 1 to 15. When using two machines, the time to build index

<sup>1</sup> GBDI Arboretum is a library C++ that implements different metric access methods (MAM) (cf. <http://www.gbdi.icmc.usp.br/old/arboretum>).



**Fig. 3.** Elapsed time for building indexes for each of the three collections

for the datasets, is 40,144 s, 41,847 s and 42,547 s, respectively. In comparison with 15 machines, we achieve a speed-up factor for all three datasets by reducing their indexing time to 7000 s. Recalling that sending the leaf nodes to client machines is done with the principle of load balancing between machines.

We observed a logical breakdown of CPU time beside the number of machine. We also noticed a logical increase compared to the complexity of the query while increasing the parameter  $k$  as well the intrinsic dimension. We found that this new approach is able to index up to twenty million objects distributed over fifteen clusters, which was our goal. We recall that the choice of destination clusters between machines during the construction of the index was done in a way that the distribution of objects was almost balanced on all machines. Note that communication between client machines and also the exchanges of responses plays a very important role in improving response time, so the effectiveness of our index.

## 5 Conclusion

In this paper, we have clarified some methods of indexing in metric spaces. Everything is put on a taxonomy of most existing indexing techniques in the literature. Afterwards, we presented a study (GHB-tree), a proposition that was inspired from GH-tree. This technique is incremental, not dependent on a defined data type, and especially easy to construct the index. GHB-tree is a peer-to-peer system supporting similarity search in metric spaces. Compared with the available state-of-the-art, our method significantly improves the query retrieval process. Extensive experimental results show that this improvement, for  $k$ NN queries, increases directly proportional with the size of the network, adding ground to our scalability claims.

## References

1. Chen, L., Gao, Y., Li, X., Jensen, C.S., Chen, G.: Efficient metric indexing for similarity search and similarity joins. *IEEE Trans. Knowl. Data Eng.* **29**, 556–571 (2017). Print ISSN 1041-4347



2. Gonzaga, A.S., Cordeiro, R.L.F.: A new division operator to handle complex objects in very large relational datasets. In: EDBT (2017)
3. Wan, Y., Liu, X., Wu, Y.: CD-Tree: a clustering-based dynamic indexing and retrieval approach. *Intell. Data Anal.* **21**, 243–261 (2017)
4. Kouahla, Z., Martinez, J.: A new intersection tree for content-based image retrieval. In: 10th International Workshop on Content-Based Multimedia Indexing, CBMI 2012, Annecy, France, 27–29 June 2012, pp. 1–6 (2012). <https://doi.org/10.1109/CBMI.2012.6269793>
5. Ciaccia, P., Patella, M., Zezula, P.: M-tree: an efficient access method for similarity search in metric spaces. In: Proceedings of the 23rd VLDB International Conference, pp. 426–435 (1997)
6. Almeida, J., Valle, E., Torres, R.S., Leite, N.J.: DAHC-tree: an effective index for approximate search in high-dimensional metric spaces. *J. Inf. Data Manag.* **1**(3), 375–390 (2010)
7. Pola, I.R.V., Traina, A.J.M., Traina Jr., C., Kaster, D.S.: Improving metric access methods with bucket files. In: Amato, G., Connor, R., Falchi, F., Gennaro, C. (eds.) SISAP 2015. LNCS, vol. 9371, pp. 65–76. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-25087-8\\_6](https://doi.org/10.1007/978-3-319-25087-8_6)
8. Ooi, B.C.: Spatial kd-tree: a data structure for geographic database. In: Schek, H.J., Schlageter, G. (eds.) *Datenbanksysteme in Büro, Technik und Wissenschaft. Informatik-Fachberichte*, vol. 136, pp. 247–258. Springer, Heidelberg (1987). [https://doi.org/10.1007/978-3-642-72617-0\\_17](https://doi.org/10.1007/978-3-642-72617-0_17)
9. Burkhard, W.A., Keller, R.M.: Some approaches to best-match file searching. *Commun. ACM* **16**(4), 230–236 (1973)
10. Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: Proceedings of the 4th Annual in ACM-SIAM Symposium on Discrete Algorithms, pp. 311–321 (1993)
11. Nielsen, F.: Bregman vantage point trees for efficient nearest neighbor queries. In: Proceedings of Multimedia and Exp (ICME). IEEE (2009)
12. Fu, A.W., Chan, P.M., Cheung, Y.-L., Moon, Y.S.: Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *VLDB J. Very Large Data Bases* **9**, 154–173 (2012)
13. Curtin, R.R.: Faster dual-tree traversal for nearest neighbor search. In: Amato, G., Connor, R., Falchi, F., Gennaro, C. (eds.) SISAP 2015. LNCS, vol. 9371, pp. 77–89. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-25087-8\\_7](https://doi.org/10.1007/978-3-319-25087-8_7)
14. Arroyuelo, D.: A dynamic pivoting algorithm based on spatial approximation indexes. In: Traina, A.J.M., Traina, C., Cordeiro, R.L.F. (eds.) SISAP 2014. LNCS, vol. 8821, pp. 70–81. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-11988-5\\_7](https://doi.org/10.1007/978-3-319-11988-5_7)
15. Pagh, R., Silvestri, F., Sivertsen, J., Skala, M.: Approximate furthest neighbor in high dimensions. In: Amato, G., Connor, R., Falchi, F., Gennaro, C. (eds.) SISAP 2015. LNCS, vol. 9371, pp. 3–14. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-25087-8\\_1](https://doi.org/10.1007/978-3-319-25087-8_1)
16. Ulhmann, J.K.: Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.* **40**, 175–179 (1991)
17. Carélo, C.C.M., Pola, I.R.V., Ciferri, R.R., Traina, A.J.M., Traina Jr., C., de Aguiar Ciferri, C.D.: Slicing the metric space to provide quick indexing of complex data in the main memory. *Inf. Syst.* **36**, 79–98 (2011)