Improving metric access methods with bucket files

Improving Metric Access Methods with Bucket Files

Ives R.V. Pola[2](✉), Agma J.M. Traina[1], Caetano Traina Jr.[1], and Daniel S. Kaster[2](✉)

[1] University of São Paulo, São Carlos, Brazil
{agma,caetano}@icmc.usp.br
[2] University of Londrina, Londrina, Brazil
{ives,dskaster}@uel.br

Abstract. Modern applications deal with complex data, where retrieval by similarity plays an important role in most of them. Complex data whose primary comparison mechanisms are similarity predicates are usually immersed in metric spaces. Metric Access Methods (MAMs) exploit the metric space properties to divide the metric space into regions and conquer efficiency on the processing of similarity queries, like range and k-nearest neighbor queries.

Existing MAM use homogeneous data structures to improve query execution, pursuing the same techniques employed by traditional methods developed to retrieve scalar and multidimensional data. In this paper, we combine hashing and hierarchical ball partitioning approaches to achieve a hybrid index that is tuned to improve similarity queries targeting complex data sets, with search algorithms that reduce total execution time by aggressively reducing the number of distance calculations. We applied our technique in the Slim-tree and performed experiments over real data sets showing that the proposed technique is able to reduce the execution time of both range and k-nearest queries to at least half of the Slim-tree. Moreover, this technique is general to be applied over many existing MAM.

1 Introduction

The existing Data Base Management Systems (DBMS) were originally developed to store and retrieve data represented in numeric and short character strings domains. They are not able to efficiently manage the complex data handled by current applications, such as multimedia data, georeferenced data, time series, genetic sequences, scientific simulations, etc. The main reason precluding those data to be appropriately managed by current DBMSs is because their internal structures require the data domains to comply with the ordering relationship (OR) properties, that is, they require that every data element from a domain can be compared by the $<, \leq, >$ and $\geq$ operators. To manage complex data even the equality comparison operators $=$ and $\neq$ are almost useless, because identity seldom occurs (or is not worth pursuing) when retrieving complex data. To query complex data, comparing by similarity is the most important operation [6].

G. Amato et al. (Eds.): SISAP 2015, LNCS 9371, pp. 65–76, 2015.
DOI: 10.1007/978-3-319-25087-8_6

Similarity search is the most frequent abstraction to compare complex data, based on the concept of proximity to represent similarity embodied in the mathematical concept of metric spaces [8]. The development of the Metric Access Methods (MAMs), also known as distance-based index structures, provides adequate techniques to retrieve complex data, once they are based solely on the distances (similarities) between pairs of elements in a data set. Evaluating (dis)similarity using a distance function is desirable when the data can be represented in metric spaces. Formally, a metric space is a pair $\langle \mathbb{S}, d \rangle$, where $\mathbb{S}$ is the data domain and $d : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{R}^+$ is the distance function, or metric, that holds the following properties for any $s_1, s_2, s_3 \in \mathbb{S}$:

- Identity ($d(s_1, s_2) = 0 \rightarrow s_1 = s_2$);
- Symmetry ($d(s_1, s_2) = d(s_2, s_1)$);
- Non-negativity ($0 < d(s_1, s_2) < \infty$, $s_1 \neq s_2$) and
- Triangular inequality ($d(s_1, s_2) \leq d(s_1, s_3) + d(s_3, s_2)$).

Given a set S in a complex domain $\mathbb{S}$, a similarity query returns a result set $\mathsf{T}_R = \{s_i \in S\}$ that meet a given similarity criterion, expressed through a reference element $s_q \in \mathbb{S}$. For example, for image databases one may ask for images that are similar to a given one, according to a specific criterion. There are two main types of similarity queries: the range and the k-nearest neighbor queries.

There are two broad classes of access methods that exploit the properties of metric spaces: those based on the hierarchical division of the space based on ball-shaped regions centered at one element, and those based on pivots sets, which can be implemented as a hierarchy or as hash tables. Dynamic MAMs have had special attention by academy and industry as they do not degrade with updates.

In this paper we propose the Bucket-Slim-Tree (BST), a MAM based both on hash and on ball partitioning, aiming at reducing the number of distance evaluations required to answer a similarity query. BST employs the dynamic MAM Slim-tree as a kind of hash function mapping to buckets delimited by a fixed radius. Such organization allows reducing the overlap among regions improving the search performance in a great extent. Experiments over real data sets reported in the paper show that BST demands less than half of the distance calculations and of the execution to perform similarity queries when compared to the original Slim-tree, in the best results.

The rest of the paper has the following outline. Section 2 discusses the background and existing works related to this one. Section 3 presents the proposed data structure, the Bucket-Slim-Tree. Section 4 shows experiments performed to demonstrate the improvement of the proposed structure. Finally, Section 5 concludes for this work.

2 Background

Metric access methods use only the distances between elements to prune further comparisons in subsets of the elements during search. Pruning techniques require the algorithms to store distances to take advantage of the metric properties and/or of statistics from the distance distribution over the data space. The usual pruning techniques use lower bounds of distances derived from the triangular inequality property. Another approach is to store the minimum and the maximum distances within a group of elements to help discarding entire regions during search algorithm execution.

Many indexing structures were developed exploiting those concepts, such as the Geometric Near Access Tree (GNAT) of Brin [3], leading to the class called Voronoi-based MAMs. The EGNAT [10] is a dynamic variation of GNAT, which provides a mechanism to store elements on disk by creating buckets on the leaf nodes and also enables deletion. Another approach, the ball decomposition scheme, partitions the dataset based on distances from a distinguished element called a Vantage Point (VP), thus creating the so-called VP-tree [14]. The VP-tree construction process is based on finding the median element of a sorted sample list of the elements, which leads to a recursive tree creation. Other disk-based MAM have been proposed based on the VP-tree, such as the MVP-tree [2], where multiple vantage points are used.

The BP-tree (Ball-and-Plane tree) [1] is constructed by recursively dividing the data set into compact, low-overlap clusters. It is static and was designed to deal with high dimensional data, where a data distribution analysis is used to search in clusters.

Several dynamic, disk-based MAMs were proposed in the literature[13][4]. A disk-based MAM requires the structure to hold many elements per node, in order to decrease the number of disk accesses. They employ a bottom up strategy to construct the trees, assuring the creation of balanced trees. The M-tree [4] was the first of such trees proposed, followed by the Slim-tree [13], which includes the Slim-down algorithm to reduce node overlaps. The OMNI concept [11] increases the pruning power of search operations using a few elements strategically positioned as pivots, the foci set. These methods store distances among the elements and the pivots, so the triangular inequality property can be used to prune nodes and reduce the number of sub-tree accesses.

Most of the indexing structures presented above are based on ball partitioning or pivot-based structures. Hashing, the “key to address” mapping, is the basis for D-Index [5] and SH [7]. The LAESA algorithm [9] is a pivot table that uses a matrix of distances between all pairs of pivots selected from the dataset. When processing queries, it sequentially process the entire distance matrix (or parts of it in multiple passes), pruning by using the triangle inequality property. But, the internal cost of LAESA can be so high that it can be equivalent to perform a sequential search when indexing high dimension datasets at low cost metrics [12].

Both ball and hash based methods have particular advantages that can be combined to achieve better metric structures. The way Ball-based MAM partitions the metric space leads to a better organization of the data structure, so every resulting partition of the metric space groups similar elements. However, the best dynamic approaches produce regions that overlap, imposing to the search algorithm to visit many regions. The pivot partitioning methods are affected by the pivots selection policy and how they are combined to prune regions. Hash-based methods usually partition the data into subsets that are addressed later to answer the queries.

Our proposal is innovative as it exploits the best properties from ball-based and from pivot-based methods. Specifically, our method Bucket-Slim-Tree (BST) uses the Slim-tree as a hash function to search within a bucket file to improve performance. BST merges the usage of buckets of elements with the Slim-tree, enabling to explore properties from both structures that results in an overall reduced consumption of computational resources.

3 The Bucket-Slim-Tree

The Bucket-Slim-Tree (BST) is composed of a Slim-tree and a set of buckets pointed by the Slim-tree leaf nodes. The Slim-tree acts as a hash function that during query answering determines which bucket should be visited next. The basic structure of a BST is shown in Figure 1. Each element in a leaf node has a pointer to its respective bucket. Although each Slim-tree node have a limited capacity, each bucket is (theoretically) limitless.

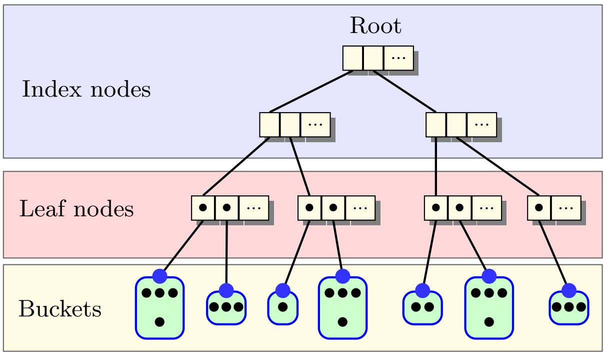
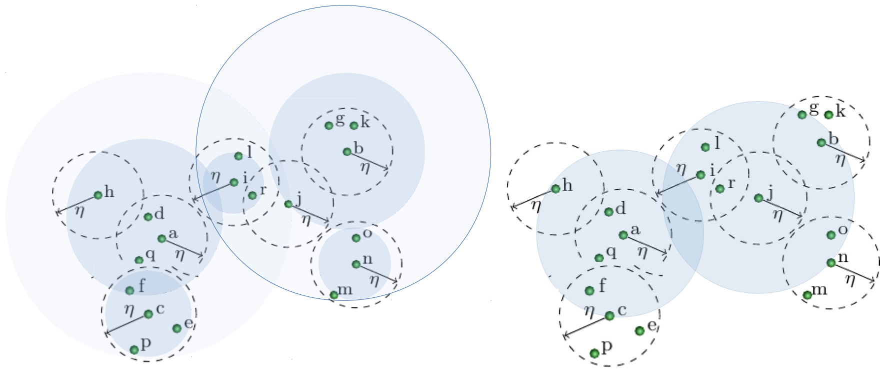


Fig. 1. Structure of a Bucket-Slim-Tree.

The elements indexed in the index and leaf nodes are considered search keys, and act like pivots in a hash structure. Inserting new elements or answering queries require to traverse the tree structure using the search keys to determine which leaf nodes will have their buckets accessed for inspection. The query result will be composed of keys from the Slim-tree and also of elements stored in the corresponding buckets that match the search criteria.

3.1 The Structure of the Buckets

A BST is constructed for a specific bucket radius η, given beforehand. A bucket $B^{\eta}(bc_i)$ represents a ball of radius η in the metric space, whose center is the



(a) Slim-tree (b) BST

Fig. 2. Ball partitioning comparison between regular Slim-tree and BST.

element bc_i stored in a Slim-tree leaf node pointing to that bucket. Thus, a bucket stores elements s_i such that $\forall s_i \in B^{\eta}(bc_i) : d(s_i, bc_i) \leq \eta$. Each element s_i that belongs to a bucket does not belong to other buckets.

For example, consider Figure 2. In Figure 2(a) it shows a set of points in $\mathbb{R}^2$ indexed in a regular Slim-tree with four levels. But, in Figure 2(b) it shows the same set of points with buckets of a fixed radius η centered at elements $\{a, b, c, h, i, j, n\}$, indexed on a BST. As it can be seen, there can be empty buckets, such as $B^{\eta}(h)$, and elements that are covered by more than one bucket are stored in only one bucket, such as element r. Note that, comparing both figures, using buckets reduces the overall covering radius of the index elements, reducing the overlap in the structure.

The bucket radius plays an important role in the performance of the Bucket-Slim-Tree. Setting too large η values will result in large buckets, thus creating long subsets of elements to be analyzed during queries. This degenerates into sequential scans in the buckets and few key elements to filter the buckets in the tree. Furthermore, the larger is the region covered by a bucket the more the overlap between sibling buckets, which leads to potentially more unnecessary accesses. On the other hand, choosing too small values for η will produce many small or empty buckets, leading the search cost to occur mostly in the Slim-tree and an added internal cost to manage the buckets. Choosing the bucket radius $\eta = 0$ the result is the Slim-tree itself, thus the Bucket-Slim-Tree can be seen as a generalization of the Slim-tree.

Next we will discuss how to build the Bucket-Slim-Tree, i.e., how to choose the keys and how to create the buckets.

3.2 Building the Bucket-Slim-Tree

The Bucket-Slim-Tree is a dynamic MAM able to be constructed either using bulk-loading or adding elements one at a time. The BST is designed to group similar elements into buckets centered at the key elements stored in the Slim-tree. As elements are added, some of them are stored in the Slim-tree leaf nodes, thus becoming keys to the buckets, and others are stored in the buckets. The way that those keys are organized in the Slim-tree affects how many buckets are necessary to answer each query. The more bucket regions overlap the more buckets need to be visited in a query. Therefore, it is important to choose an index creation policy that reduces such overlap, even if it results in a deeper tree.

Elements are added to a Bucket-Slim-Tree following Algorithm 1. When a new element s_n arrives, the basic insertion algorithm of the Slim-tree is executed to find the appropriate leaf node L_m where it would be inserted. However it is not inserted yet. Next, the buckets from the keys stored in node L_m are evaluated looking for the keys $bc_i \in L_m$ such that $d(bc_i, s_n) \leq \eta$. The new element s_n is stored in the qualifying bucket whose center is the closest to s_n, along with the distance from the bucket center. If no bucket qualifies, s_n is stored in L_m splitting the node if required, as in the regular Slim-tree insertion algorithm, and s_n becomes a new bucket center. However, the corresponding bucket is not created now – it will be created only when another element is stored in it. Once the structure is constructed, similarity queries can be performed considering η as an additional pruning radius, as is explained following.

Algorithm 1. BST:ADD(s_n)

```
Input: new element $s_n$
var candidate : bucket center that covers $s_n$
var leaf : leaf node of Slim-tree
Set *chooseSubTree* policy of Slim-tree to 'MINDIST'
Set leaf to the proper leaf node that covers $s_n$
foreach *element* $bc_i$ *in leaf* do
    if $d(bc_i, s_n) \leq \eta$ then
        Add $bc_i$ as a candidate
if *there are candidates* then
    Choose the first center $bc_i$ where $d(bc_i, s_n)$ is minimum
    Insert $s_n$ in the bucket of $bc_i$ and store $d(bc_i, s_n)$
else
    Add $s_n$ to leaf
    Split leaf if necessary
End
```

3.3 Querying the Bucket-Slim-Tree

The BST structure allows performing both range (*Rng*) and k-nearest neighbor (k-NN) similarity queries. The algorithms to answer those queries visit both the nodes in the tree and the buckets. For both query types, radius η must be taken into account to correctly prune subtrees at each index level. As radius η is a fixed value defined beforehand of the BST construction, it is possible to avoid the need to adjust each region formed in the Slim-tree during construction by adding η to every query radius.

An example of a range query $Rng(s_q, \xi)$ is shown in Figure 3, considering a two-dimensional set of points using the Euclidean distance. The element ds_1 shown in the figure is a representative in an index node, so it is also stored in a leaf node L_m. Elements bc_1 to bc_4 are elements stored in leaf node L_m of the Slim-tree. Thus, node L_m has five elements stored: $\{ds_1, bc_1, bc_2, bc_3, bc_4\}$, and each one is the center of a bucket.

Each bucket is shown as a dashed ball in Figure 3, which represents the space region whose corresponding elements (e.g. $s_1, s_2 \ldots$) will be stored. Thus, to avoid pruning valid buckets (like the one centered at bc_1 in the subtree centered at ds_1) the query radius ξ must be adjusted to $\xi + \eta$. In Figure 3, this corresponds to change the query ball drawn in solid line centered at s_q to the one drawn in dotted line. In this example, only the bucket centered at element bc_1 must be evaluated, adding element s_1 to the result. The same idea applies to the k-nearest neighbor query, which requires to enlarge the dynamic radius by η.

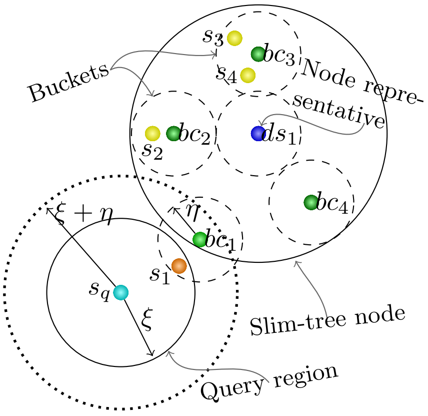


Fig. 3. Querying a Bucket-Slim-Tree.

The similarity query algorithms use the triangular inequality to prune subtrees as in the original Slim-tree and also inside each bucket. In Figure 3 example, instead of calculating every distance $d(s_q, bc_i), bc_i \in L_m$, just evaluate the lower bound of the required distance using the triangular inequality, avoiding a calculation whenever $d(s_q, bc_i) \geq |d(s_q, ds_1) - d(ds_1, bc_i)|$. Notice that the values $d(s_q, ds_i)$ are already stored in BST, and that $d(s_q, ds_i)$ is evaluated only once for each leaf node. Thus, assuming the pruning radius $r_p = \eta + \xi$, whenever $|d(s_q, ds_i) - d(ds_1, bc_i)| > r_p + \eta$ then bucket bc_i can be safely pruned without evaluating $d(s_q, bc_i)$.

The steps to evaluate a range query$Rng(s_q, \xi)$ is shown in Algorithm 2, where s_q is the query center and ξ is the query radius. To traverse the tree, the algorithm evaluates the index nodes using both η and ξ to qualify the subtrees that must be visited. To process the leaf nodes, only the radius η is used.

The procedure of a k-nearest neighbors query k-$NN(s_q, k)$ in the BST is analogous to the range query, but now we update the result list maintaning k elements and updating the active radius. the technique of a shrinking active radius starts with a value larger than the dataset diameter (or infinity), and reduces when the ongoing result list achieves k elements and updates.

4 Experiments

In this section we show experiments to evaluate the proposed index structure, the Bucket-Slim-Tree. We compare it with the original Slim-Tree, using different values for bucket radius (η). The experiments show that using the bucket-based

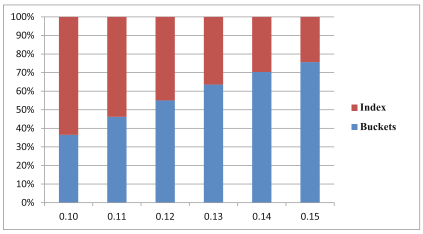
Algorithm 2. *RangeQuery*($s_q, \xi, root$)

```
Input: Query center $s_q$, Query radius $\xi$, Slim-tree $root$
if $root$ is index node then
    foreach $ds_i \in root$ do
        //Evaluate if the triangular inequality allows pruning;
        if $|d(ds_i, root) - d(root, s_q)| > \eta + \xi + ds_i.Radius$ then
            Prune subtree of $ds_i$;
        foreach element $ds_i$ not pruned do
            if $d(ds_i, s_q) \leq \eta + \xi + ds_i.Radius$ then
                RangeQuery($s_q$, $\xi$, $ds_i$.Subtree);
if $root$ is a leaf node then
    foreach $bc_i \in root$ do
        if $d(bc_i, s_q) \leq \xi$ then
            add $bc_i$ to result;
        //Evaluate if the bucket can be pruned
        if $d(bc_i, s_q) \leq \eta + \xi$ then
            foreach $s_i \in B(bc_i)$ do
                //Evaluate if the triangular inequality allows pruning;
                if $|d(s_i, root) - d(root, s_q)| \leq \xi$ then
                    if $d(s_i, s_q) \leq \xi$ then
                        add $s_i$ to result;
```

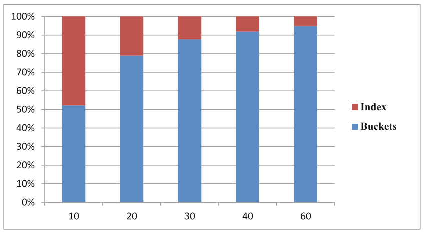
approach increases the query answering performance being up to twice faster than slim-tree. They also show how the query performance is affected when different bucket sizes are employed.

We used three datasets for the experiments. The `Corel` Dataset consists of 10 thousand color histograms in a 32 dimension space extracted from an image set, using the L_1 distance. The `USCities` Dataset consists of the latitude and longitude coordinates of 25,376 cities in the USA, using the great-circle distance modified to return distances in kilometers. The `HCimages` Dataset was obtained from a collection of 500,000 DICOM images from the Ribeirão Preto Medical School Clinics Hospital of the University of São Paulo (HCFMRP-USP). From each image, we extracted a 256-bin grayscale normalized histogram. All the experiments were performed in a machine with a Intel Core i7 920 processor with 8 Gb RAM of memory.

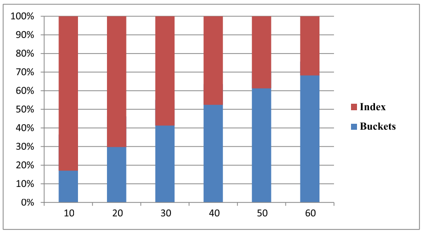
The first experiment measured how the buckets are filled with elements according its radius η. The plots in Figure 4 show the percentage of element distributed among the slim-tree and the buckets, with bucket radius η varying from 0.10 to 0.15 for Corel (Figure 4(a)), from 10 Km to 60 Km for USCities (Figure 4(b)) and from 10 to 60 for HCimages (Figure 4(c)). The plots for Corel show that, as the bucket radius increases, the percentile of elements stored in the buckets increases from 36% to 75%. Similar behavior occur in the plot for HCimages, but in this case the number of elements in the buckets increase slower. Exemplifying the case where a high value for η produces dense buckets, the plot for USCities shows that as the radius η we increases from 10 to 60 Km for the `USCities` dataset, the number of elements stored in the buckets reaches almost 95%, meaning that almost all elements are



(a) Corel Dataset



(b) USCities Dataset.



(c) HCimages Dataset.

Fig. 4. Distribution of elements in the Bucket-Slim-Tree components varying the bucket radius η.

stored in the buckets, probably degenerating the structure, where queries would sequentially scan dense buckets.

The performance of a BST depends on the chosen value for η. This value changes for different datasets and should be set close to the frequently used radius on range queries in order to achieve good results. An initial value can be given by a percentage of the value of the dataset maximum radius, or estimating the mean distance from all elements in the dataset. All experiments were performed using different values of η for both range and k-nearest neighbor queries. As previously noted, the BST uses a modified Slim-tree with the *mindist* policy for the *ChooseSubTree* algorithm. For comparison purposes, we also evaluated the results if it is employed a Slim-tree with the usual *minoccup* policy. Every query was performed 500 times with the same radius ξ or k but different centers, in order to evaluate the average of the number of performed distance calculations and the total time spent.

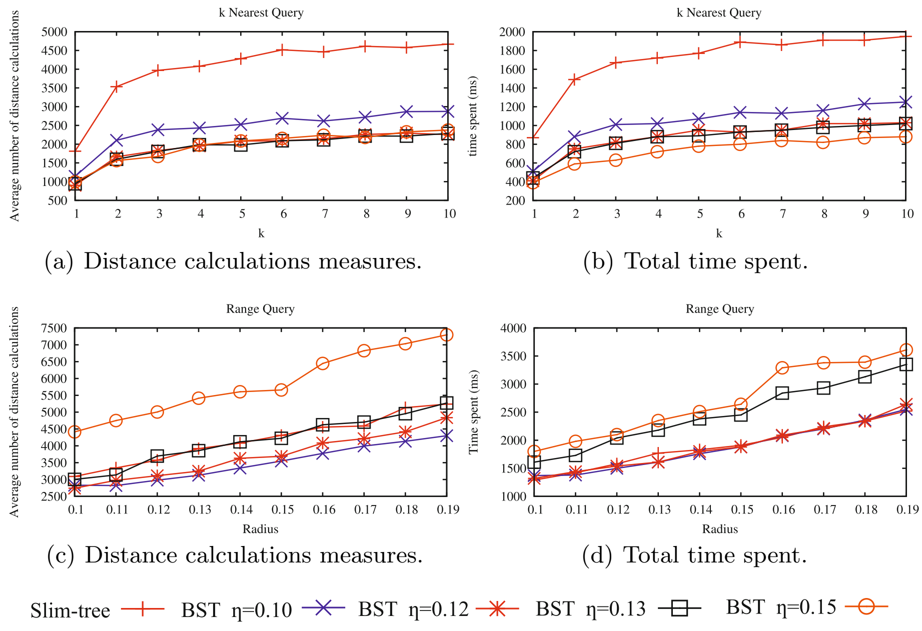
The plots in Figure 5 show the results of measuring the performance for both types of queries using the `Corel` Dataset. They show that in the beginning, as the value of η increases, the query performance increases. However, if η becomes too high, the performance is decreased, as shown in Figure 5(c) when $\eta > 0.12$. This is because buckets become larger and the sequential scans inside each bucket spend more time.

The plots in Figures 6 show the performance measurements for both types of queries using the `USCities` dataset, obtained using η set to 10, 20 and 40 Km. As it can be noticed, all configurations lead to BST with a performance better than that of a slim-tree for all queries, where the value of 20 Km produced the best one. It is important to notice that for $\eta = 40$km, the performance was worse than for $\eta = 20$km. This is because for radius larger than $\eta = 20$km, the number of elements in the buckets tends to increase too much, as shown in Figure 4(b).

The plots in Figures 7 show the performance results for queries using the `HCimages` dataset, using η with the values 10, 20 and 30. As this dataset has a high dimensionality, any variation of the radius will strongly change the covering of elements, as expected of the curse of the high dimensionality. From the results we can note that our technique still enhances the performance of queries when choosing η next to the query values. This is because any decrease in the index level covering radius greatly reduces the overlap on nodes.

5 Conclusion

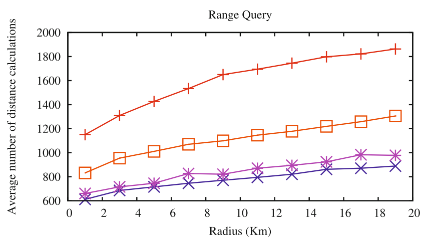
In this paper we proposed the Bucket-Slim-tree (BST), a MAM based on hash and ball partitioning that aims at reducing the number of distance calculations required to answer similarity queries. The BST is composed of a slim-tree and a set of buckets assigned to each element in the Slim-tree leaf nodes. The slim-tree acts as a hash function which maps the stored elements to the buckets that will be visited during search. The leaf nodes contain all key elements associated with a bucket of radius η, and all of them must be compared to the query element during query executions.



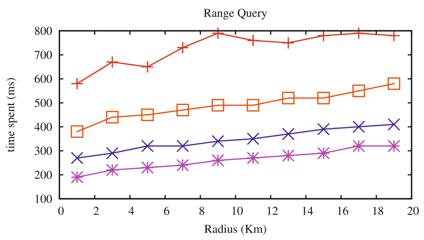
(a) Distance calculations measures. (b) Total time spent.

(c) Distance calculations measures. (d) Total time spent.

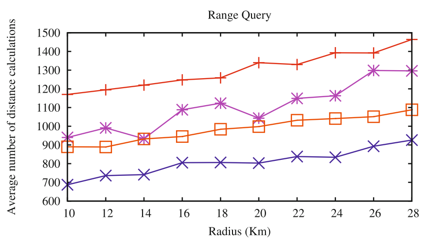
Fig. 5. Results using the `Corel` dataset indexed in a slim-tree and BSTs with $\eta = 0.10$, 0.12, 0.13 and 0.15. (a) Number of distance calculations for *k-NN* queries; (b) Time spent for *k-NN* queries; (c) Number of distance calculations for *Rq* queries; (d) Time spent for *Rq* queries;



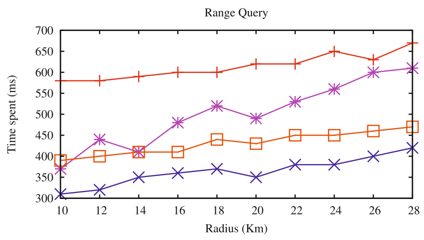
(a) Distance calculations measures.



(b) Total time spent.



(c) Distance calculations measures.



(d) Total time spent.

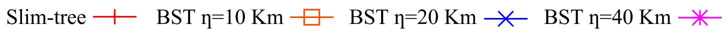
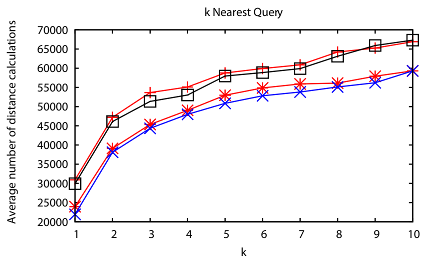
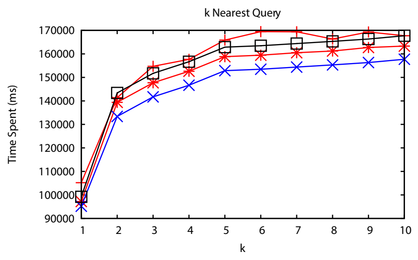


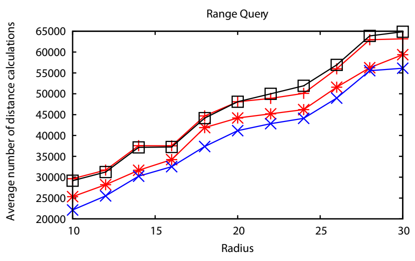
Fig. 6. Results using the `USCities` dataset. (a) and (b): Nearest Neighbor query evaluation; (c) and (d): Range query evaluation.



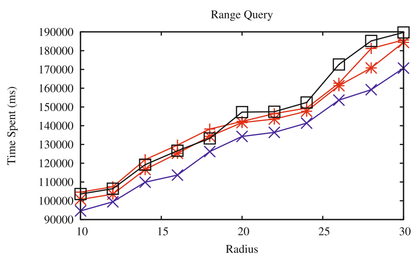
(a) Distance calculations measures.



(b) Total time spent.



(c) Distance calculations measures.



(d) Total time spent.

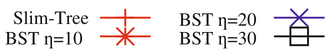


Fig. 7. Results using the `HCimages` dataset. (a) and (b): Nearest Neighbor query evaluation; (c) and (d): Range query evaluation.

Experiments performed over real data sets show that the proposed MAM was able to reduce up to half the execution time of both range and k-nearest queries, reducing also the number of distances calculations under different values of η.

Acknowledgments. This research has been partially suported by CAPES, CNPq and by FAPESP.

References

1. Almeida, J., Torres, R.d.S., Leite, N.J.: Bp-tree: an efficient index for similarity search in high-dimensional metric spaces. In: Proceedings of the 19th ACM International Conference on Information and Knowledge Management, CIKM 2010, pp. 1365–1368. ACM, New York (2010)
2. Bozkaya, T., Özsoyoglu, Z.M.: Distance-based indexing for high-dimensional metric spaces. In: ACM SIGMOD International Conference on Management of Data, Tucson, AZ, pp. 357–368. ACM Press (1997)
3. Brin, S.: Near neighbor search in large metric spaces. In: Dayal, U., Gray, P.M.D., Nishio, S. (eds.) International Conference on Very Large Databases (VLDB), break pp. 574–584. Morgan Kaufmann, Zurich (1995)
4. Ciaccia, P, Patella, M., Rabitti, F., Zezula, P.: Indexing metric spaces with m-tree. In: Atti del Quinto Convegno Nazionale SEBD, Verona, Italy, pp. 67–86 (1997)
5. Dohnal, V., Gennaro, C., Savino, P., Zezula, P.: D-index: Distance searching index for metric data sets. Multimedia Tools and Applications Journal (MTAJ) **21**(1), 9–33 (2003)
6. Faloutsos, C.: Indexing of multimedia data. In: Multimedia Databases in Perspective, pp. 219–245. Springer Verlag (1997)
7. Gennaro, C., Savino, P., Zezula, P.: Similarity search in metric databases through hashing. In: 3rd International Workshop on Multimedia Information Retrieval, Ottawa, Canada, pp. 1–5 (2001)
8. Kelley, J.L.: General Topology. Springer (1955)
9. Micó, L., Oncina, J., Vidal, E.: A new version of the nearest-neighbor approximating and eliminating search (aesa) with linear processing-time and memory requirements. Pattern Recognition Letters **15**, 9–17 (1994)
10. Navarro, G., Uribe-Paredes, R.: Fully dynamic metric access methods based on hyperplane partitioning. Inf. Syst. **36**, 734–747 (2011)
11. Santos Filho, R.F., Traina, A.J.M., Traina Jr., C., Faloutsos, C.: Similarity search without tears: the omni family of all-purpose access methods. In: IEEE International Conference on Data Engineering (ICDE), Heidelberg, Germany, pp. 623–630. IEEE Computer Society (2001)
12. Skopal, T.: Where are you heading, metric access methods?: a provocative survey. In: Proceedings of the Third International Conference on SImilarity Search and APplications, SISAP 2010, pp. 13–21. ACM, New York (2010)
13. Traina Jr, C., Traina, A.J.M., Faloutsos, C., Seeger, B.: Fast indexing and visualization of metric datasets using slim-trees. IEEE Transactions on Knowledge and Data Engineering (TKDE) **14**(2), 244–260 (2002)
14. Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms (SODA), Austin, TX, pp. 311–321 (1993)