A new dynamic, secondary-memory metric index

Rodrigo Paredes[1][0000−0002−9943−2510], Nora Reyes[2][0000−0001−7630−3161], Karina Figueroa[3][0000−0002−4680−5950], and Manuel Hoffhein[4]

[1] Universidad Finis Terrae, Santiago, Chile
[2] Universidad Nacional de San Luis, San Luis, Argentina
[3] Universidad Michoacana, Morelia, Mexico
[4] Universidad de Talca, Curicó, Chile
raparede@uft.cl, nreyes@unsl.edu.ar,
karina@fismat.umich.mx, manuel.hoffhein@audienceview.com

Abstract. Metric space searching addresses the problem of efficient similarity searching in many applications. Although promising, the metric space approach is still immature in several aspects that are well established in traditional databases. Particularly, most indexing schemes are not dynamic, that is, few of them tolerate insertion of elements at reasonable cost over an existing index with none or mild performance degrading; and even less of them work efficiently in secondary memory. The *List of Clusters* (LC) is a competitive index in main memory. We introduce a new dynamic, secondary-memory variant of the LC. Our new index handles well the secondary memory scenario and is competitive with the state of the art, becoming a useful alternative in a wide range of database applications. Also, our ideas are applicable to other secondary-memory indexes, where it is possible to control the disk page occupation.

1 Introduction

Metric space searching addresses the problem of efficient similarity searching in many applications, such as multimedia databases, text retrieval, function prediction, and many others. The similarity between objects can be measured with a distance function, usually considered expensive to compute, defined by experts in a specific data domain. Therefore, all those applications share some common characteristics: a finite *dataset* of elements belonging to a *metric space*, where a *distance function* is used to assess similarity. When a *similarity query* is posed to this dataset, it consists basically in, given a new element of the space called the *query* q, looking for elements of the dataset that are similar enough to q. In order to efficiently answer queries, the dataset is preprocessed so as to build an *index* that reduces query time. This metric space approach is becoming widely popular [14, 16] and several indexing methods have flourished [2, 7, 14], but it is still immature in several aspects that are well established in traditional databases.

Most of the existing indexes are *static*: they need to known all the dataset beforehand to build the index and once it is built, adding more elements to the dataset, or removing an element from it, requires an expensive updating of the index. There are few indexes that tolerate some insertions, but their quality degrade and require periodic rebuildings. Others tolerate deletions with the same quality degradation problem. Therefore, there are very few *dynamic* indexes [10]. In addition, these few dynamic indexes are designed thinking of both objects and the index itself will be maintained in main memory [13].

However, there are many interesting databases for similarity searching where the objects are so large that they must stay on disk; or the objects are so many that the index itself cannot fit in main memory. In this case, although the similarity computation can be expensive (e.g., taking milliseconds of CPU time), we cannot disregard the costs of I/O operations. Thus, from the few dynamic indexes, even less of them work well in secondary memory: most of them need the data structure in main memory in order to operate efficiently.

Even though it may be acceptable a static scheme for some applications, many relevant ones do require dynamic capabilities. For instance, in digital libraries and archival systems, versioned and historical databases, and several other scenarios, it would be enough to support insertions because objects are never updated nor deleted. Moreover, when the erasing operation is unfrequent, one can resort to lazy deletion, that is, marking the object as erased, which allows excluding it in a search, but use it in the index as necessary.

Metric space searching becomes more difficult as the so-called intrinsic dimension of the space increases [9]. The practical effect of raising the intrinsic dimensionality is to make objects look more similar, degrading search performance. Thus, we are interested in dynamic secondary-memory indexes for medium- and high-dimensional spaces. The existing alternatives for this scenario are clustering-based indices. The best known metric index from this family is the *M-tree* [3]. Several others followed, for example the *EGNAT* [12], the *D-index* [5], the *PM-tree* [15], the *DSAT** and *DSAT+* [11], and the *DLC* and the *DSC* [13].

In this article, we propose a new clustering-based dynamic index for secondary memory built on the Dynamic Set of Clusters [13], which, in turn, is inspired by a simple static structure that performs very well on high-dimensional spaces, the *List of Clusters (LC)* [1]. We designed a new dynamic version of the *DSC* that works in secondary memory, which we call *Buffered On Line Dynamic Set of Clusters (BOLDSC)*. This structure requires a very low number of I/Os and has a balanced performance in searches and updates. We focus on range searches, however the structures are capable of answering nearest neighbor queries by inheriting the corresponding main-memory algorithms [4].

The rest of the paper is structured as follows. Basic concepts and previous work are presented in Section 2 and 3, respectively. Section 4 describes our proposal and Section 5 evaluates it. Finally, Section 6 shows our conclusions.

2 Basic Concepts

Let $\mathbb{U}$ be a universe of *objects*, with a *distance function* $d : \mathbb{U} \times \mathbb{U} \longrightarrow \mathbb{R}^+ \cup \{0\}$ defined among them. The function d satisfies the three properties that make $(\mathbb{U}, d)$ a *metric space*: *strict positiveness*, *symmetry*, and *triangle inequality*. We handle a finite *dataset* $S \subseteq \mathbb{U}$, which is a subset of the universe of objects and can be preprocessed to build an index. Later, given a new object from the universe $\mathbb{U}$ (a *query* $q \in \mathbb{U}$), we must retrieve all similar elements found in the dataset. There are two basic kinds of queries: *range* and *k-nearest neighbor* queries. We focus this work on range queries, where given $q \in \mathbb{U}$ and $r \in \mathbb{R}^+$, we need to retrieve all elements of S within distance r to q.

An algorithm aims to establish some structure or index over the dataset $\mathbb{S}$; then, when a query object q is given, the algorithm uses this structure to speed up the response time. Of course, in order to traverse through the index some distances computations are needed. This process obtains a set of non-discarded objects and q is compared with all these objects to answer the similarity query.

The distance is assumed to be expensive to compute. However, when we work in secondary memory, the search complexity must also consider the I/O time; other components such as CPU time for side computations can be disregarded. The I/O time is composed of the number of disk pages read and written; we call B the size of the disk page. Given a dataset of $|S| = n$ objects of total size N and disk page size B, queries can be trivially answered by performing n distance evaluations and $\lceil N/B \rceil$ I/Os. The goal of an index is to preprocess the dataset so as to answer queries with as few distance evaluations and I/Os as possible.

In terms of memory usage, one considers the extra memory required by the index on top of the data and, in the case of secondary memory, the disk page utilization, that is, the average fraction of the disk pages that is used.

In a dynamic scenario, the set S may undergo insertions and deletions, and the index must be updated accordingly for the subsequent queries. It is also possible to start with an empty index and build it by successive insertions.

In some dynamic scenarios, deletions do not occur. In others, they are sufficiently rare to permit a simple approach to handle them: one marks the deleted objects and exclude them from the output of queries; the index is rebuilt when the proportion of deleted objects exceeds a certain threshold. However, when deletions are frequent, or when the objects are large and retaining them for the sole purpose of indexing is unacceptable, we need to address the more challenging case, where deletions must be physically executed.

3 Previous Works

We briefly describe previous works related to our proposal: List of Clusters (LC) [1], Dynamic Spatial Approximation Tree (DSAT) [10], and Dynamic Set of Clusters (DSC) [13].

3.1 List of Clusters

We briefly recall the *List of Clusters* (LC) [1]. It splits the space into zones (or "clusters"). Each zone has a center c, a radius r_c, and stores the *internal* objects $I = \{x \in S,\ d(x, c) \leq r_c\}$, which are and inside a previous zone.

The construction proceeds by choosing c and r_c, computing I, and then building the rest of the list with the remaining elements $E = S \setminus I$. Many alternatives to select centers and radii are considered [1], finding experimentally that the best performance is achieved when the zones have a fixed number of elements m (and r_c is defined accordingly for each c), and the next center c is selected as the element that maximizes the distance sum to the previously chosen centers. The brute force algorithm for constructing the list takes $O(n^2/m)$ time.

A range query (q, r) visits the list zone by zone. We first compute $d(q, c)$, and report c if $d(q, c) \leq r$. Then, if $d(q, c) - r_c \leq r$ we search exhaustively the set of internal elements I. The rest of the list is processed only if $r_c \leq d(q, c) + r$.

3.2 Dynamic Spatial Approximation Tree

The *Dynamic Spatial Approximation Tree* (DSAT) is built incrementally, via insertions. The maximum tree node arity is a parameter. Each tree node a stores a timestamp of its insertion, $time(a)$, its covering radius, $R(a)$, which is the maximum distance to any element in its subtree, and its set of children $N(a)$.

To insert a new element x, its place is sought starting at the tree root and moving to the neighbor closest to x, updating $R(a)$ in the way. We finally insert x as a new (leaf) child of a if (1) x is closer to a than to any $b \in N(a)$, and (2) the arity of a, $|N(a)|$, is not already maximal. Neighbors are stored left to right in increasing timestamp order. So, the parent is always older than its children.

At search time the insertion process is replicated. We proceed as if we were inserting q keeping in mind that relevant elements x may be at $d(q, x) \leq r$.

3.3 Dynamic Set of Clusters

The *Dynamic Set of Clusters* (DSC) is a hybrid structure. On the one hand, it stores the objects in a set of clusters, each of them represented by its centers, in secondary-memory. On the other hand, it uses a DSAT [10] to manage the cluster centers in main memory, it supports searches, insertions, and deletions,, and manages changes in centers of cluster.

The DSC stores the objects I of a cluster in a single disk page, so that the cluster retrieval incurs only one disk page read. Therefore, it uses clusters of fixed size m, which is chosen according to the disk page size B. For each cluster C the index stores (1) the center object $c = center(C)$; (2) its covering radius $r_c = cr(C)$ (the maximum distance between c and any object in the cluster); (3) the number of elements in the cluster, $|I| = \#(C)$; and (4) the objects in the cluster, $I = cluster(C)$, together with the distances $d(x, c)$ for each $x \in I$. In order to reduce I/Os, it maintains components (1), (2), and (3) in main memory. Thus it is possible to determine whether a zone has to be scanned without reading data from disk. The cluster objects and their distances to the center (component (4)) are maintained in the corresponding disk page.

In main memory, for each cluster C, the DSAT stores $c = center(C)$, $cr(C)$, $\#(C)$, and a value $CRZ(c)$, the maximum distance between c and any element stored in the cluster of any center in the subtree of c.

The DSC starts empty and is built by successive insertions. The first arrived element becomes the center of the first cluster, and hence the root of the DSAT of centers. From then on, we apply a general insertion mechanism described next.

The only part of the index that is always up to date in secondary memory are the clusters and their centers. Besides, these are located in consecutive disk pages. Therefore, if any problem occurs and the DSAT of centers is lost or corrupted, it can be rebuilt with a sequential scan of clusters.

4 Our Proposal

As we mentioned, the DSC obtains good search performance and can be built via insertions. However, as the clusters are conformed as insertions arrive, it cannot ensure the cluster sharpness nor the 50% of disk page occupancy. Hence, we

designed a new dynamic, secondary-memory index, called *Buffered On Line Dynamic Set of Clusters* (BOLDSC), which chooses centers and clusters in a better way, and thus it improves search performance and insertions of new elements.

The BOLDSC key idea is that not each insertion produces a write operation on disk. To do so, we maintain in memory a buffer of elements, which is a temporal bag, and when it is full we choose a center c and the elements of its cluster. Only in this moment, this new cluster (including c) is written on disk and its center along some additional information is incorporated into the DSAT of centers. As DSC does, BOLDSC stores the cluster objects I in a single disk page, so that a cluster retrieval incurs only one disk page read. Besides, it also uses clusters of fixed size m, chosen according to the disk page size B.

For each cluster C we store in a disk page (1) its center object $c = center(C)$; (2) its covering radius $r_c = cr(C)$ (the maximum distance between c and any object in the cluster); and (3) the objects in the cluster, $I = cluster(C)$ along with their distance to c. As it is aforementioned, allocated in main memory, the DSAT of centers maintains the main information of each cluster, as DSC does.

In order to support at any time efficient searches, we use a simple pivot approximation to index elements inside the buffer, in addition to the DSC. Therefore, the BOLDSC actually maintains in main memory three structures: the DSAT with the information of clusters stored in secondary memory, the temporal bag of elements, and the set of current pivots to index the bag.

For each cluster C, the cluster BOLDSC maintains into the DSAT the same information that DSC: its center c, its covering radius cr, the address of its disk page on secondary memory $\#p$, and the $CRZ(c)$, the covering radius of the DSAT subtree which root is c. In the buffer we maintain the elements which are not yet part of any cluster, and for each object in the bag we store its distances to the current set of pivots. The maximum number of elements b that the buffer can maintain on main memory is a multiple d of the number of elements that fit in one cluster. Thus, if according to the disk page size B one cluster can hold a center and m objects inside the cluster, $b = d \cdot (m + 1)$.

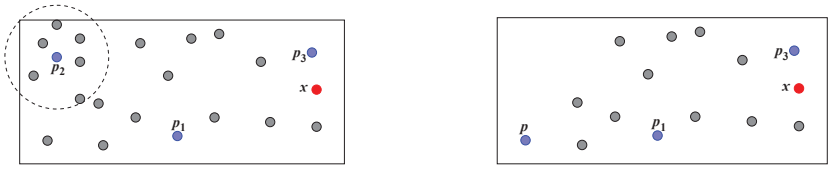
4.1 Insertion

When a new object arrives, we have to consider in which of the following situations the temporal bag is: vspace*-4mm

- It already contains $b - 1$ elements, or
- It has less than $b - 1$ elements.

When a new element x arrives and the temporal bag contains less than $b - 1$ elements, we only have to put x in the bag and calculate and store all the distances between x and each pivot p_i in the current set of pivots $\mathcal{P}$.

On the other hand, when a new element x arrives and the bag becomes full (that is, it now has $b - 1$ elements), we select the pivot p_i out of the set $\mathcal{P} = \{p_1, p_2, \ldots, p_s\}$, such that its covering radius that encloses its m nearest neighbors inside the bag is the lowest one. Then, we extract p_i and its m nearest neighbors from the bag to form a cluster. So, the information of this cluster is added into the DSAT of centers, and the complete cluster is written in one



(a) Insertion of x. (b) After insertion of x.

Fig. 1. Example of the temporal bag with the insertion of a new element x, and after this insertion occurs.

disk page. We also remove p_i from the set $\mathcal{P}$ and the distances between the remaining bag objects and p_i. Then, we select as a new pivot the element p of the bag having the maximum accumulated distance to the remaining pivots in $\mathcal{P}$. Next, we calculate the distance between p and all the elements that stay inside the bag. This way, we maintain a dynamic set of pivots that fits to the current set of objects in the bag.

Therefore, for each $m + 1$ insertions we create a new cluster and we need to write this cluster in a disk page. So, the cost of the write operation of disk is amortized between $m + 1$ insertions. Besides, we have pages with a 100% of occupancy and the clusters on these pages are fairly compact.

Fig. 1 depicts an example. The insertion of x fills up the bag capacity and then the region around pivot p_2 will be extracted to form the new cluster. We consider in this case that the maximum cluster size is $m = 6$, Fig. 1(a). The next situation of the bag is showed in Fig. 1(b). The element p is selected as the new pivot because it is the farthest element from both p_1 and p_3 together.

Moreover, we can ensure that all the expensive disk operations will be optimized because the clusters have a 100% of page occupancy and the file will have as less pages as possible. Therefore, if the file F is located trough f disk cylinders, the disk read head will have a mean movement of $f/3$ to seek on any random page of F, while f is the minimum number of disk cylinders to store F.

4.2 Searches

To solve a range query, we need to traverse both the bag and the clusters in secondary memory. We use the pivots in the bag as usual [2, 7, 14]. Next, we traverse the DSAT of centers to determine which clusters must be reviewed. Of course, for each zone intersecting the query ball, we access the clusters in secondary memory, avoiding irrelevant clusters. Finally, the range query answer is the union of both outcomes, from the bag and the clusters.

5 Experimental Results

For the empirical evaluation of the BOLDSC, we first consider two widely different metric spaces from the SISAP Metric Library (`www.sisap.org`) [6]. They are not very large, but are useful to tune the index and make some design decisions based on their rough performance.

– *Words,* a dictionary of 69,069 English words. We use the *edit distance*, that is, the minimum number of character insertions, deletions and substitutions

needed to make two strings equal. This distance is useful in text retrieval to cope with spelling, typing and optical character recognition (OCR) errors.
– *Color histograms,* a set of 112,682 8-D color histograms (112-dimensional vectors) from an image database[5]. Any quadratic form can be used as a distance; so, we choose Euclidean as the simplest meaningful distance.

In all cases, we built the indices with 90% of the points and used the other 10% (randomly chosen) as queries. All our results are averaged over 10 index constructions using different permutations of the datasets.

Words have a discrete distance, so we used radii 1 to 4, which retrieve on average 0.00003%, 0.00037%, 0.00326% and 0.01757% of the dataset, respectively. For the color histograms, we consider range queries retrieving on average 0.01%, 0.1% and 1% of the dataset. This corresponds to radii 0.051768, 0.082514 and 0.131163. The same queries are used for all the experiments on the same datasets. As said, given the existence of range-optimal algorithms for k-nearest neighbor searching [7, 8], we have not considered these search experiments separately.

The disk page size is $B = 4$KB. All the tree data structures cache the tree root in main memory. All the indices are built by successive insertions. We evaluate the behavior of BOLDSC using several values of its parameter d (how many times the number of elements of one cluster), but we only show the values: 10, 50, 100, 200, 300, 400. These values are representative enough of all the values evaluated. For example, $d = 100$ means that the bag has 400 KB of elements.

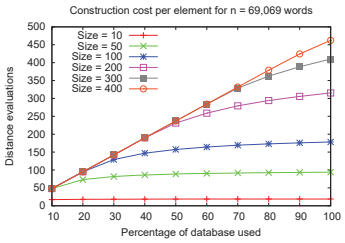
We depict the construction costs measured in distance evaluations and number of I/O operations (Fig. 2) needed for each insertion, considering both metric spaces. As it can be seen, the bigger the bag, the more distance evaluations are needed for each insertion. This is because there is one pivot for each page in the bag with which the new element has to be compared. But, the number of I/O operations (it only writes disk pages) decreases as the bag has more elements. Actually BOLDSC needs very few average disk operations for each insertion.

Fig. 3 shows the search costs of BOLDSC measured in distance evaluations and number of I/O operations, for both metric spaces varying the bag size. Obviously, as BOLDSC has more elements within the bag, the clusters are more compact. As it can be noticed, as the BOLDSC bag size increases the search costs decreases, both in distance evaluations and number of I/O operations. However, it is important to consider that as the metric spaces considered are not so large, almost all the database elements fit in the bag when its size is large.

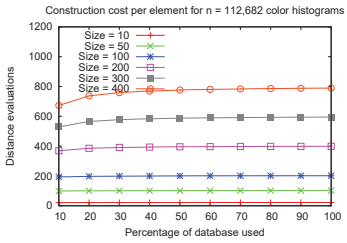
Finally, we show a brief validation of our proposal against another secondary memory metric index. We compare our BOLDSC using bag size of 50, 100, and 200 disk pages, because these values have a reasonable compromise between memory space used and performance for both metric spaces considered. Since DSC has the best trade-off between construction and search costs [13], we use it considering the best options informed for each metric space in [13].

Fig. 4 depicts the comparison of construction costs in distance evaluations and number of I/O operations. As can be expected, BOLDSC requires significantly less I/O operations, although the number of distances evaluations does

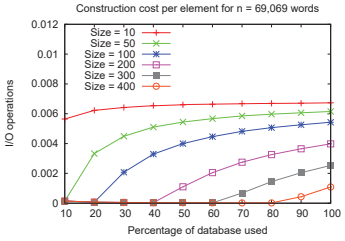
[5] At `http://www.dbs.informatik.uni-muenchen.de/~seidl/DATA/histo112.112682.gz`



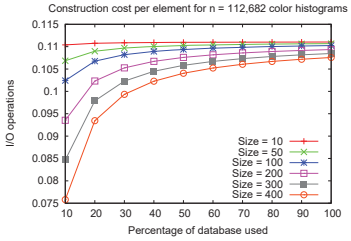
(a) Distance evaluations Words.



(b) Distance evaluations Histograms.

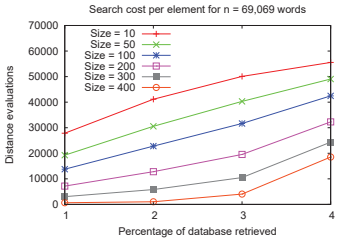


(c) I/O operations Words.

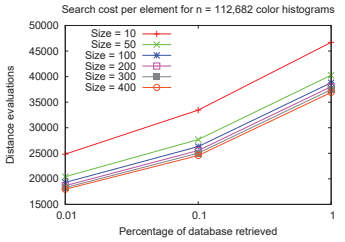


(d) I/O operations Histograms.

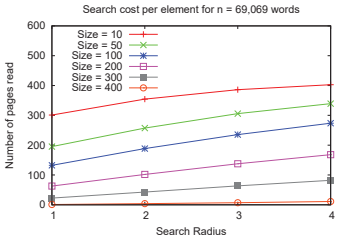
Fig. 2. BOLDSC construction costs varying the bag size.



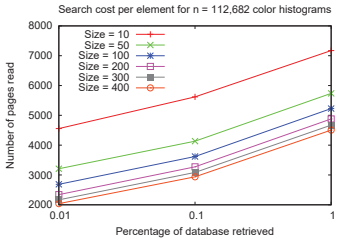
(a) Distance evaluations Words.



(b) Distance evaluations Histograms.



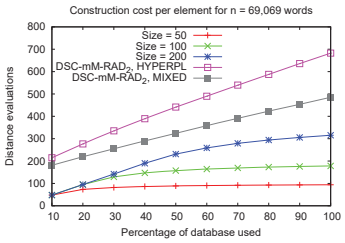
(c) I/O operations Words.



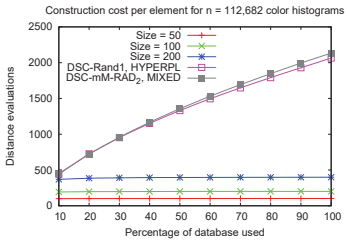
(d) I/O operations Histograms.

Fig. 3. BOLDSC search costs varying the bag size.

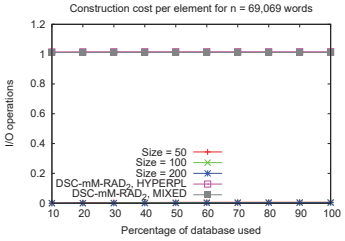
not differ that much. The comparison of search costs is showed in Fig. 5, considering distance evaluations and number of I/O operations. In the space of strings, BOLDSC performs better, and this is inverted for the space of color histograms.



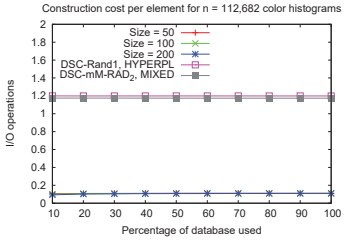
(a) Distance evaluations Words.



(b) Distance evaluations Histograms.

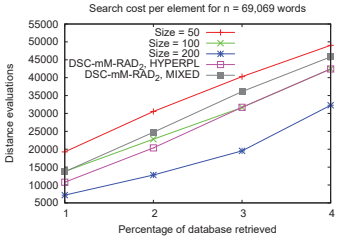


(c) I/O operations Words.

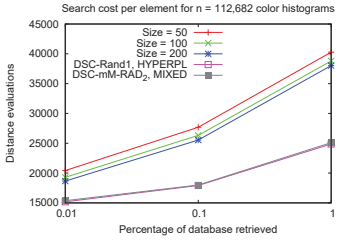


(d) I/O operations Histograms.

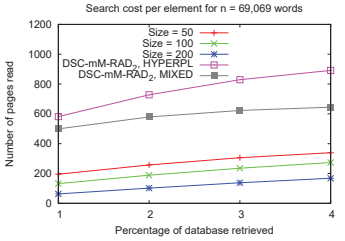
Fig. 4. Comparison of construction costs of BOLDSC with DSC.



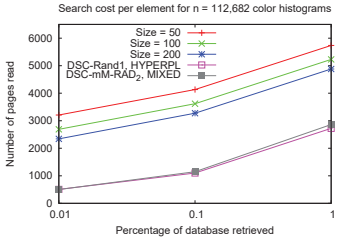
(a) Distance evaluations Words.



(b) Distance evaluations Histograms.



(c) I/O operations Words.



(d) I/O operations Histograms.

Fig. 5. Comparison of search costs of BOLDSC with DSC.

6 Conclusions

We have presented the *Buffered On Line Dynamic Set of Clusters* (BOLDSC). Our BOLDSC is inspired on the List of Clusters, and it is specially adapted

to the dynamic and secondary-memory scenario. The key idea is using a buffer to both choose the clusters in a better way and delay the write operation on disk until we have a good cluster candidate. To improve the search efficiency, we include within the index a DSAT to manage the cluster centers.

The empirical evaluation included in this work has show that BOLDSC sports a promising performance. We verify that the BOLDSC has a 100% disk page utilization and its construction is efficient int I/O operations, but has to be improved in distance evaluations. Also, the BOLDSC search performance is competitive with respect the state of the art.

As future work, we are considering to study the dependence on the number of pivots to index the buffer and use some massive datasets.

References

1. Chávez, E., Navarro, G.: A compact space decomposition for effective metric indexing. Pattern Recognition Letters **26**(9), 1363–1376 (2005)
2. Chávez, E., Navarro, G., Baeza-Yates, R., Marroquín, J.: Searching in metric spaces. ACM Computing Surveys **33**(3), 273–321 (2001)
3. Ciaccia, P., Patella, M., Zezula, P.: M-tree: an efficient access method for similarity search in metric spaces. In: Proc. 23rd VLDB. pp. 426–435 (1997)
4. Costa, V.G., Marín, M., Reyes, N.: Parallel query processing on distributed clustering indexes. J. Discrete Algorithms **7**(1), 3–17 (2009). https://doi.org/10.1016/j.jda.2008.09.010
5. Dohnal, V., Gennaro, C., Savino, P., Zezula, P.: D-index: Distance searching index for metric data sets. Multimedia Tools and Applications **21**(1), 9–33 (2003)
6. Figueroa, K., Navarro, G., Chávez, E.: Metric spaces library (2007), available at `http://www.sisap.org/Metric_Space_Library.html`
7. Hjaltason, G., Samet, H.: Index-driven similarity search in metric spaces. ACM Transactions on Database Systems **28**(4), 517–580 (2003)
8. Hjaltason, G., Samet, H.: Incremental Similarity Search in Multimedia Databases. Computer science technical report series, Computer Vision Laboratory, Center for Automation Research, University of Maryland (2000)
9. Navarro, G., Paredes, R., Reyes, N., Bustos, C.: An empirical evaluation of intrinsic dimension estimators. Information Systems **64**, 206–218 (March 2017)
10. Navarro, G., Reyes, N.: Dynamic spatial approximation trees. ACM Journal of Experimental Algorithmics **12**, article 1.5 (2009)
11. Navarro, G., Reyes, N.: Dynamic spatial approximation trees for massive data. In: Proc. 2nd SISAP. pp. 81–88 (2009)
12. Navarro, G., Uribe, R.: Fully dynamic metric access methods based on hyperplane partitioning. Information Systems **36**(4), 734–747 (2011)
13. Navarro, G., Reyes, N.: New dynamic metric indices for secondary memory. Inf. Syst. **59**, 48–78 (2016). https://doi.org/10.1016/j.is.2016.03.009
14. Samet, H.: Foundations of Multidimensional and Metric Data Structures. Morgan Kaufmann Publishers Inc. (2005)
15. Skopal, T., Pokorný, J., Snásel, V.: PM-tree: Pivoting metric tree for similarity search in multimedia databases. In: ADBIS (Local Proceedings) (2004)
16. Zezula, P., Amato, G., Dohnal, V., Batko, M.: Similarity Search: The Metric Space Approach, Advances in Database Systems, vol. 32. Springer (2006)