Dynamic List of Clusters in Secondary Memory

Gonzalo Navarro[1]* and Nora Reyes[2]

[1] Center of Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Chile, gnavarro@dcc.uchile.cl

[2] Departamento de Informática, Universidad Nacional de San Luis, Argentina, nreyes@unsl.edu.ar

Abstract. We introduce a dynamic and secondary-memory-based variant of the *List of Clusters*, which is shown to be competitive with the literature, especially on higher-dimensional spaces, where it outperforms the *M-tree* in searches and I/Os used for insertions. The basic principles of our design are applicable to other secondary-memory structures.

1 Introduction

The metric space approach has become popular in recent years [2, 14, 16, 6] and a large number of indexing methods have flourished. Most of the research, however, is still in the stage of static solutions that work in main memory. Static indexes have to be rebuilt from scracth when the set of indexed objects undergoes insertions or deletions. In-memory indexes can handle only small datasets, suffering serious performance degradations when the objects reside on disk. Most real-life database applications require indexes able to work on disk and to support insertions and deletions of objects interleaved with the queries.

To date, there exist only a few indexing structures supporting dynamism and designed for secondary memory. Some are based on so-called pivots [5, 8, 13], some on hierarchical clustering [3, 11, 12], and some on combinations [4, 15].

A further challenge is that the metric spaces arising in many applications are intrinsically high-dimensional, that is, the histogram of distances is concentrated. Pivot-based indexes are known to perform well on low-dimensional spaces, whereas hierarchical clustering indexes handle medium dimensions better. A simple structure that has shown to perform well on higher-dimensional spaces is the *List of Clusters (LC)* [1], but it is a static in-memory structure. There is a dynamic version of LC, named *Recursive List of Clusters (RLC)* [9], but it is also designed to work in main memory.

In this paper we introduce a dynamic and secondary-memory variant of the List of Clusters, aiming at higher-dimensional spaces. Our secondary memory version, *DLC*, retains the good features of the *LC*, and in addition performs well on secondary memory. In this paper we focus on handling searches and insertions (thus enabling incremental construction), leaving deletions for future work (these are usually handled with lazy deletion mechanisms). Our experimental

* Funded with Basal Funds FB0001, Conicyt, Chile.

comparisons show that our structures need little extra space, achieve very good disk page utilization, and are competitive with state-of-the-art alternatives. For example, compared to the *M-tree* [3], the best known alternative structure, the *DLC* is more efficient at searches. For insertions, the *DLC* performs fewer I/Os, but more distance computations. Overall, the *DLC* turns out to be a practical and easy-to-implement index that fits several practical scenarios.

2 Basic Concepts

Let $\mathbb{U}$ be a universe of *objects*, with a nonnegative *distance function* $d : \mathbb{U} \times \mathbb{U} \longrightarrow \mathbb{R}^+$ defined among them. This distance function satisfies the three axioms 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 (a *query* $q \in \mathbb{U}$), we must retrieve all similar elements found in the dataset. There are two basic kinds of queries: *range query* and *k-nearest neighbor queries.* We focus this work on range queries, where given $q \in U$ and $r > 0$, we need to retrieve all elements of S within distance r to q.

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.

The distance is assumed to be expensive to compute. However, when we work in secondary memory, the complexity of the search must also consider the I/O time; other components such as CPU time for side computations can usually 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.

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.

3 List of Clusters

We briefly recall the list of clusters (LC) [1]. The LC splits the space into zones (or "clusters"). Each zone has a center c and a radius r_c, and it stores the *internal* objects $I = \{x \in S,\ d(x, c) \leq r_c\}$, which are at distance at most r_c from c.

The construction proceeds by choosing c and r_c, computing I, and then building the rest of the list with the remaining elements, $E = S - I$. 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 when the next center c is selected as the element that maximizes the distance sum to the centers previously chosen. 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 $r_c \leq d(q, c) + r$.

4 Our Proposal

In this section we introduce the *DLC*. We base our index on the *LC* [1], and also use some ideas from the *M-tree* [3]. The challenge is to maintain a disk layout that minimizes both distance computations and I/Os, and achieves a good disk page utilization.

4.1 Structure

We store the objects I of a cluster in a single disk page, so that the retrieval of the cluster incurs only one disk page read. Therefore, we use clusters of fixed size m, which is chosen according to the disk page size B.[3]

For each cluster C the index stores (1) the center object $c = \mathit{center}(C)$; (2) its covering radius $r_c = \mathit{cr}(C)$ (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, $I = \mathit{cluster}(C)$, together with the distances $d(x, c)$ for each $x \in I$. In order to reduce I/Os, we will maintain components (1), (2) and (3) in main memory, that is, one object and a few numbers per cluster. The cluster objects and their distances to the center (component (4)) will be maintained in the corresponding disk page.

Unlike in the static *LC*, the dynamic structure will not guarantee that I contains *all* the objects that are within distance r_c to c, but only that all the objects in I are within distance r_c to c. This makes maintenance much simpler, at the cost of having to consider, in principle, all the zones in each query.

The structure starts empty and is built by successive insertions. The first arrived element becomes the center of the first cluster, and from then on we apply a general insertion mechanism described next.

4.2 Insertions

To insert a new object x we must locate the most suitable cluster for accommodating it. The structure of the cluster might be improved by the insertion of x. Finally, if the cluster overflows upon the insertion, it must be split somehow.

Two orthogonal criteria determine which is the "most suitable" cluster. On one hand, choosing the cluster whose center is closest to x yields more compact zones, which are then less likely to be read from disk and scanned at query time. On the other hand, choosing clusters with lower disk page occupancy yields better disk usage, fewer clusters overall, and a better value for the cost of a disk page read. We consider the two following policies to choose the insertion point:

Compactness: the cluster C whose $\mathit{center}(C)$ is nearest to x is chosen. If there is a tie, we choose the one whose covering radius will increase the least. If there is still a tie, we choose the one with least elements.

[3] In some applications, the objects are large compared to disk pages, so we must relax this assumption and assume that a cluster spans a constant number of disk pages.

Occupancy: the cluster C with lowest $\#(C)$ is chosen. If there is a tie, we choose the cluster whose $center(C)$ is nearest to x, and if there is still a tie, we choose the one whose covering radius will increase the least.

As it can be noticed, to determine the cluster where the new element will be inserted it suffices with the information maintained in main memory, thus no I/Os are incurred, only distance computations between x and the cluster centers. Once the cluster C that will receive the insertion is determined, we increase $\#(C)$ in main memory and read the corresponding page from secondary memory.

Before updating the page on disk, we consider whether x would be a better center of C than $c = center(C)$: We compute $cr_x = \max\{d(x,y),\ y \in I \cup \{c\}\}$, the covering radius C would have if x were its center. If $cr_x < \max(cr(C), d(x,c))$, we set $center(C) \leftarrow x$ and $cr(C) \leftarrow cr_x$ in main memory, and write back $I \cup \{c\}$ to disk, with all the distances between elements and the (new) center recomputed. Otherwise, we leave the current $center(C)$ as is, set $cr(C) \leftarrow \max(cr(C), d(x,c))$, and write back $I \cup \{x\}$ to disk, associating distance $d(x,c)$ to x

This improvement of cluster qualities justifies our "compactness" choice of minimizing the distance $d(x, center(C))$ against, for example, choosing the center C with smallest $cr(C)$ resulting after the insertion of x: The insertion of elements into the clusters of their smallest centers will, in the long term, reduce the covering radii of the clusters.

On large databases, a sequential scan for the center most appropriate for insertion can be too expensive in terms of distance evaluations. To reduce this time, the centers stored in memory are organized in a *Dynamic Spatial Approximation Tree (DSAT)* [10], a fully-dynamic in-memory metric index that uses little extra space per element. Any change involving a center is then reflected in the DSAT. For insertions, we determine K candidate centers with a K-NN query in the DSAT and then select one of them according to the policy to choose the insertion point. We use K to be 10% of the centers.

When the cluster chosen for insertion is full, the procedure is different. We must split it into two clusters, the current one (C) and a new one (N), choose centers for both (according to a so-called "selection method") and choose which elements in the current set $\{c\} \cup cluster(C) \cup \{x\}$ stay in C and which go to N (according to a so-called "partition method"). Finally, we must update C and add N in the list of clusters (and in the DSAT) maintained in memory, and write C and N to disk. The combination of a selection and a partition method yields a *split policy*, several of which have been proposed for the *M-tree* [3].

Split Policies. The *M-tree* [3] considers various requirements for split policies: *minimum volume* refers to minimizing $cr(C)$; *minimum overlap* to minimizing the amount of overlap between two clusters (and hence the chance that a query must visit both); and *maximum balance* to minimizing the difference in number of elements. The latter is less relevant to our structure, because the LC is not a tree, but still it is important to maintain a minimum occupancy of disk pages.

The selection method may maintain the old center c and just choose a new one c' (the so-called "confirmed" strategy [3]) or it may choose two fresh centers (the "non-confirmed" strategy). The confirmed strategy reduces the splitting cost in terms of distance computations, but the non-confirmed one usually yields clusters of better quality. We use their same notation [3], adding _1 or _2 to the strategy names depending on whether the partition strategy is confirmed or not.

Random: The center(s) are chosen at random, with zero distance evaluations.

Sampling: A random sample of s objects is chosen. For each of the $\binom{s}{2}$ pairs of centers, the m elements are assigned to the closest of the two. Then, the new centers are the pair with least sum of the two covering radii. It requires $O(s^2m)$ distance computations ($O(sm)$ for the confirmed variant, where one center is always c). In our experiments we use $s = 0.1m$.

M_LB_DIST: Only for the confirmed case. The new center is the farthest one from c. As we store those distances, this requires no distance computations.

mM_RAD: Only for the non-confirmed case. It is equivalent to sampling with $s = m$, so it costs $O(m^2)$ distance computations.

M_DIST: Only for the non-confirmed case, and not used for the *M-tree*. It aims to choose as new centers a pair of elements whose distance approximates that of the farthest pair. It selects one random cluster element x, determines the farthest element y from x, and repeats the process from y, for a constant number of iterations or until the farthest distance does not increase. The last two elements considered are the centers. The cost of this method is $O(m)$ distance calculations.

Once the centers c and c' are choosen, the *M-tree* proposes two partition methods to determine the new contents of the clusters C and $C' = N$. The first yields unbalanced splits, whereas the second does not.

Hyperplane Partition: It assigns each object to its nearest center.

Balanced Partition: It starts from the current cluster elements (except the new centers) and, until assigning them all, (1) moves to C the element nearest to c, (2) moves to C' the elment nearest to C'.

A third strategy ensures a minimum occupancy fraction αm, for $0 < \alpha < 1/2$:

Mixed Partition: Use balanced partitioning for the first $2\alpha m$ elements, and then continue with hyperplane partitioning.

4.3 Range Search

Upon a search for (q, r), we determine the candidate clusters as those whose zone intersects the query ball, using the data maintained in memory. More precisely, for each C, we compute $d = d(q, center(C))$, and if $d \leq r$ we immediately report $c = center(C)$. Independently, if $d - cr(C) \leq r$, we read the cluster elements from disk and scan them. Note that, in the dynamic case, the traversal of the list cannot be stopped when $cr(C) \leq d + r$, as explained.

The scanning of a cluster also has a filtering stage: Since we store $d(x, c)$ for all $x \in cluster(C)$, we compute $d(x, q)$ explicitly only when $|d(x, q) - d(q, c)| \leq r$. Otherwise, we already know that $d(x, q) > r$ by the triangle inequality.

Finally, in order to perform a sequential pass on the disk when reading the candidate clusters, and avoid unnecessary seeks, we first sort all the candidate clusters by their disk page number before starting reading them one by one.

For lack of space we have focused on range search. Nearest neighbor search algorithms can be systematically built over range searches in an optimal way [7]. To find the k objects nearest to q, the main difference is that the set of candidate clusters must be traversed ordered by the lower-bound distances $d(q, center(C)) - cr(C)$, in order to shrink the current search radius as soon as possible, and the process stops when the currently known kth nearest neighbor is closer than the least $d(q, center(C)) - cr(C)$ value of an unexplored cluster.

5 Experimental Results

In order to give a broad picture of the performance of our index, we have selected three widely different metric spaces, all from the SISAP Metric Library (`www.sisap.org`). The disk page size used in this experiments is 4KB.

Words: a dictionary of 69,069 English words. The distance is the *edit distance*, that is, the minimum number of character insertions, deletions and substitutions needed to make two strings equal.

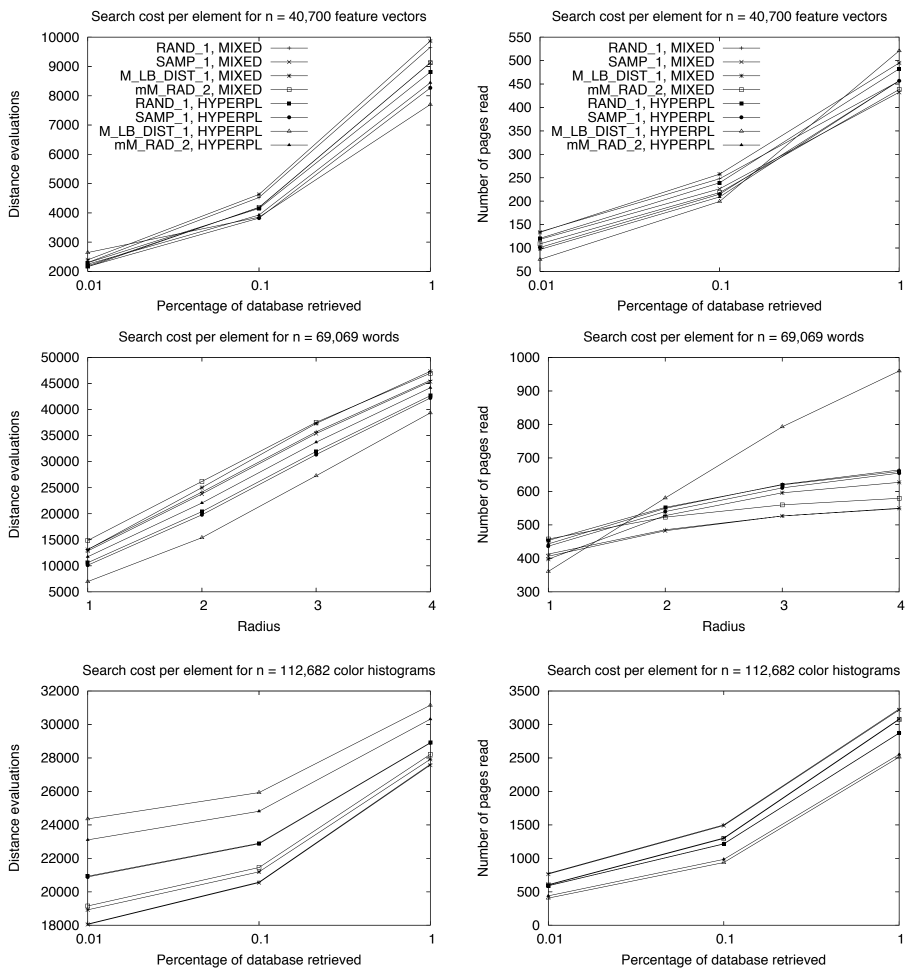
Images: 40,700 20-dimensional feature vectors, generated from NASA images, using Euclidean distance.

Histograms: 112,682 8-D color histograms (112-dimensional vectors) from an image database. Euclidean distance is used.

5.1 Search Performance

For the search experiments, we built the indexes with 90% of the elements and used the other 10% (randomly chosen) as queries. All our results are averaged over 10 index constructions using different permutations of the datasets. We have considered range queries retrieving on average 0.01%, 0.1% and 1% of the dataset. This corresponds to radii 0.605740, 0.780000 and 1.009000 for the images, and 0.051768, 0.082514 and 0.131163 for the histograms. Words have a discrete distance, so we used radii 1 to 4, which retrieved on average 0.00003%, 0.00037%, 0.00326% and 0.01757% of the dataset, respectively. The same queries were used for all the experiments on the same datasets.

For lack of space, we show the results of the best alternatives considering mainly search costs. From the point of view of searches, the best alternatives are: compactness (COMP) for the search of the insertion point, mM_RAD_2, Sampling_1 (SAMP_1), and M_LB_DIST_1 for center selection, and pure (HYPERPL) or combined with balancing (MIXED) hyperplane distribution for partitioning. As expected, the balanced partitioning obtains worse search costs than the others, because it prioritizes occupancy over compactness. The same occurs with the insertion strategy that looks for improved occupancy. Fig. 1 shows the search costs in terms of distance evaluations (1(a)) and pages read (1(b)).



(a) Distance Evaluations (b) Pages Read.

Fig. 1. Search costs for the best alternatives of *DLC*.

As it can be seen, with respect to distance evaluations, in general better costs are obtained with hyperplane distributions. For the NASA images, the best strategy of center selection depends on the radius, but a good global alternative is M_LB_DIST_1. For Words, the best alternative of center selection for all radii is M_LB_DIST_1. If we consider the number of pages read during searches for NASA images, better costs are obtained with the two versions of M_LB_DIST_1, with the distribution that ensures a minimum occupancy of disk pages (MIXED) and with hyperplane distribution. For Words, the best results are achieved with MIXED distribution and the center selection strategies SAMP_1 and Random_1 (RAND_1). Notably, the confirmed center selection policies are better than non-

Dataset	Fill ratio		Total pages used		
	DLC	*DSA+-tree*	*DLC*	*DSA+-tree*	*M-tree*
Words	34%	66%	1,288	1,536	1,608
Images	54%	67%	1,431	1,726	1,973
Histograms	45%	67%	24,922	21,136	31,791

Table 1. Average space usage for the different datasets.

confirmed ones, except for mM_RAD_2. This fact suggests that if we are not willing to spend the necessary number of distance evaluations to test all pairs of elements as centers, we should leave the old center and choose only a new one. Finally, on Histograms, the MIXED alternative is better than the HYPERPL one regarding distances, and conversely considering the number of pages read.

5.2 Comparison with Other Indexes

The *M-tree* [3] is the best-known dynamic and secondary-memory index, and its code is freely available[4]. We have used the parameter setting suggested by the authors: `SPLIT_FUNCTION = G_HYPERPL, PROMOTE_PART_FUNCTION = MIN_RAD, SECONDARY_PART_FUNCTION = MIN_RAD, RADIUS_FUNCTION = LB, MIN_UTIL = 0.2.`

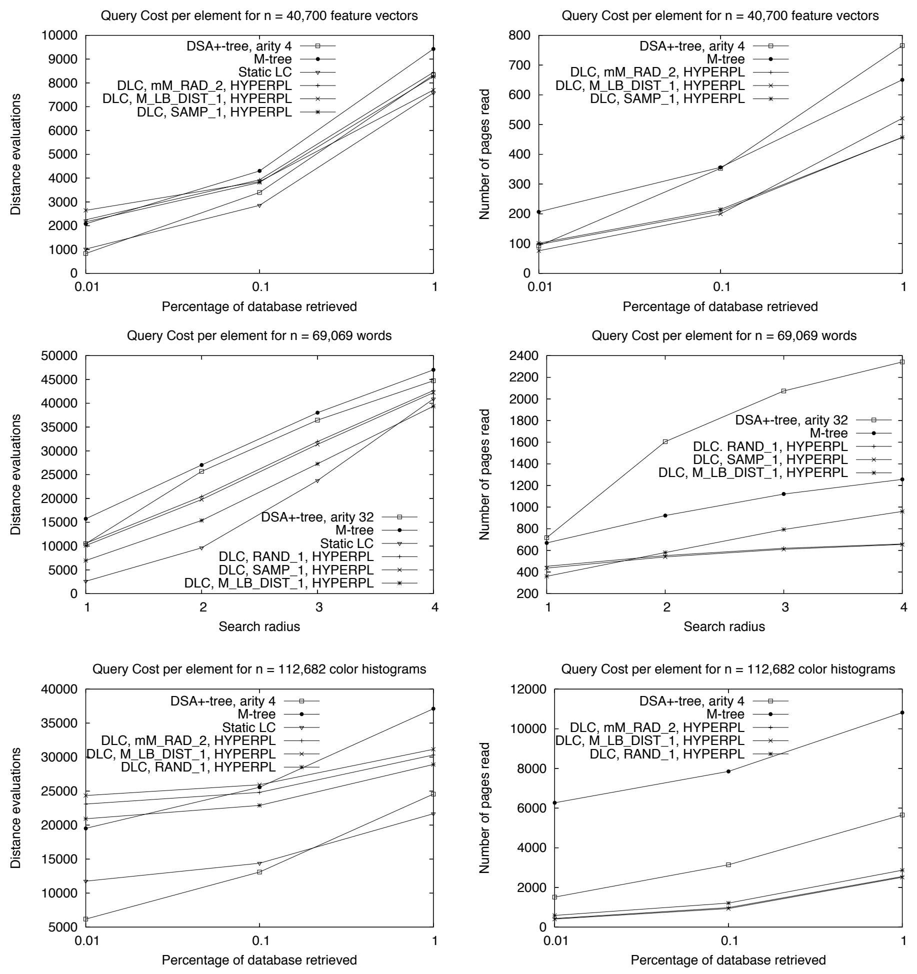
Another suitable index is the *DSA+-tree* [11]. Its only parameter is the maximum arity, for which we use the best values reported before [11] for each metric space: 4 for all the spaces except Words, where it is 32.

There are other suitable metric indexes [5, 8, 13, 4, 15], not all of which have available code. For this conference version we compare our structure with the two indexes described above, plus the static *LC*, using the same bucket size used in *DLC*, as a reference.

Table 1 shows the average disk page occupancy achieved, considering the best search alternative for the different spaces: M_LB_DIST_1 HYPERPL for Words, SAMP_1 HYPERPL for NASA images, and mM_RAD_2 HYPERPL for Histograms. The table also shows the total number of disk pages used, compared to the M-tree and the *DSA+-tree*. Our fill ratios vary depending of the space, but they are always over 30%. Although 30% occupancy is not good, even then the *DLC* is more compact than the other indexes. We remind that, by using the MIXED partition strategy, we can guarantee a minumum disk page occupancy, if desired.

Fig. 2 compares the search costs, considering distance computations (2(a)) and pages read (2(b)). In terms of distance computations, the *DSA+-tree* always takes over as the search radius grows, even outperforming the static *LC*. A larger query range makes the problem harder, equivalently to a higher dimension. For smaller radii, however, the *DSA+-tree* or the *LC* are significantly faster. In terms of disk pages read, however, the *DLC* is significantly better than the *M-tree* and the *DSA+-tree*. Only the latter gets close for small search radii on NASA images.

[4] At `http://www-db.deis.unibo.it/research/Mtree/`



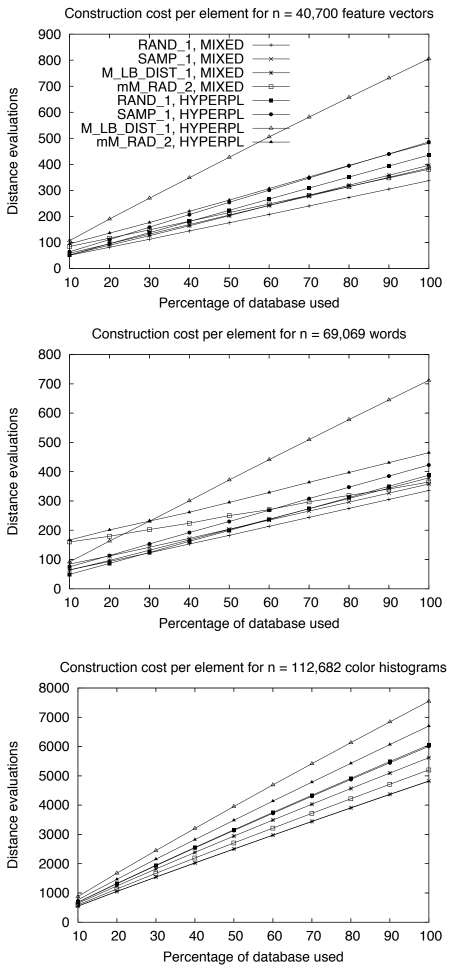
(a) Distance Evaluations (b) Pages Read.

Fig. 2. Comparison of search costs of *DLC*, *LC*, *DSA+-tree*, and *M-tree*.

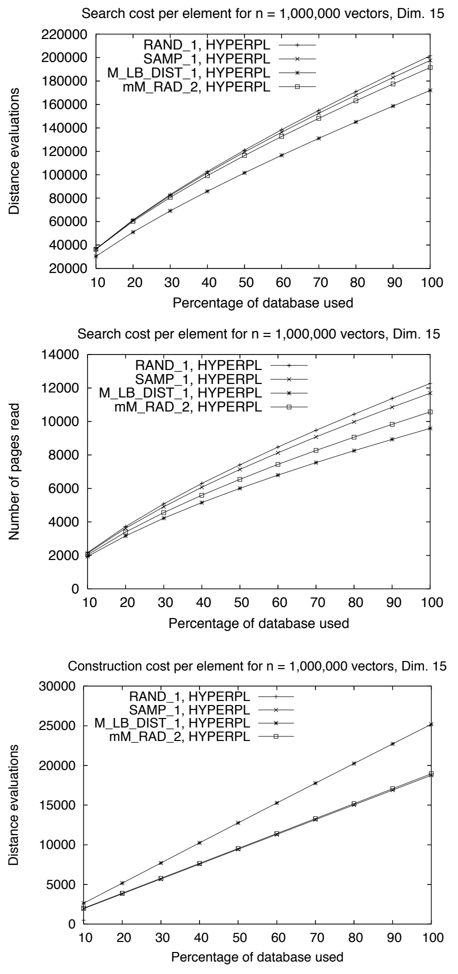
5.3 Insertion Performance

Now we analyze the insertion costs of our alternatives, and compare the best ones with previous indexes. Fig. 3(a) shows the insertion cost per element as the database grows, measured in number of distance computations. All the methods have basically the same I/O cost, 1 read and 1 write per insertion, plus a very small number equal to the average number of page splits produced, which is the inverse of the average number of objects per disk page.

Fig. 4 compares our best alternatives with previous methods, both in distance computations and I/Os. In general, *DLC* pays more distance computations for insertions than the other indexes, but it outperforms them in number of I/Os.



(a) *DLC* variants.



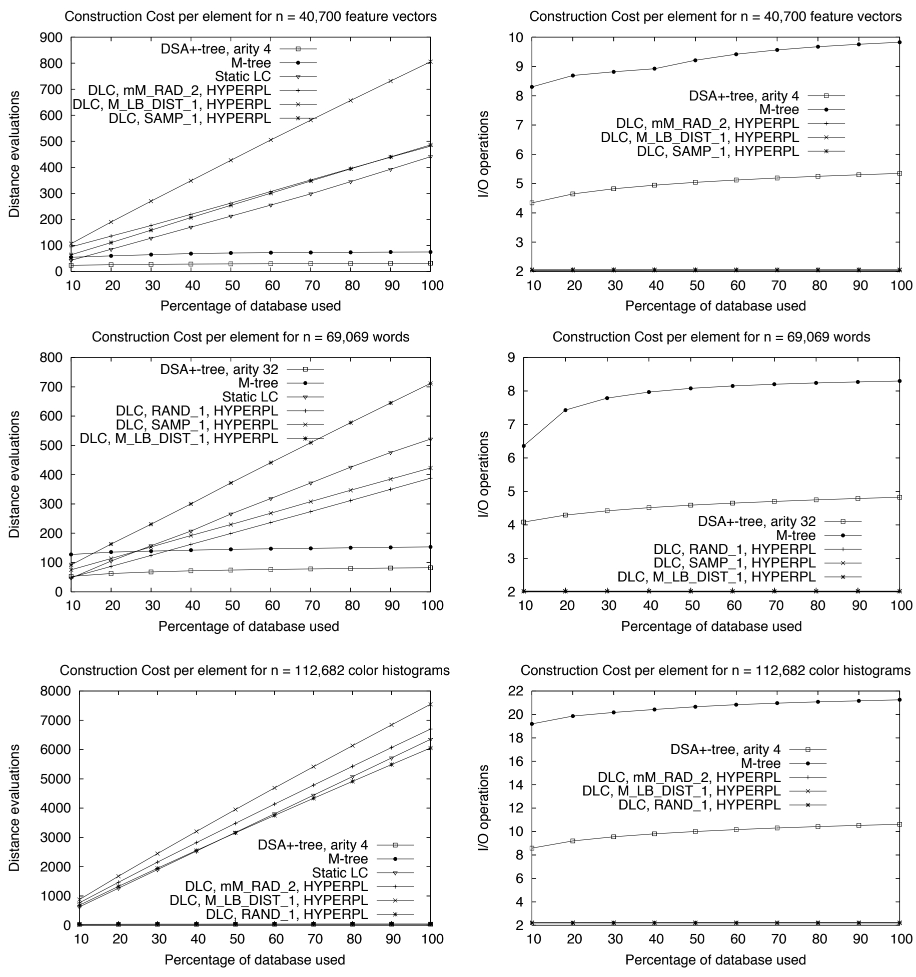
(b) Scalability Test

Fig. 3. Construction costs for best *DLC* alternatives (left) and scalability test (right).

5.4 Scalability

Fig. 3(b) shows the search costs in terms of distance evaluations, number of pages read, and construction costs (in terms of distance evaluations) on a larger synthetic dataset composed of 1,000,000 random vectors on dimension 15, uniformly distributed on the unitary hypercube.

The conclusions obtained for the smaller datasets are roughly maintained for this larger one. The fill ratio for the best searching strategy is over 30%. A more thorough study of the performance of the index on more massive scenarios is left for future work.



(a) Distance evaluations. (b) I/O operations.

Fig. 4. Construction costs of *DLC*, *LC*, *DSA+-tree*, and *M-tree*.

6 Conclusions

We have presented the *Dynamic List of Clusters (DLC)*, a dynamic and secondary-memory variant of the *List of Clusters* [1], which maintains its simplicity, low space overhead, and a good search performance in high dimensions. The *DLC*, in addition, supports efficient insertions and works in secondary memory. It achieves a reasonable disk page utilization (30% to 54%) and is competitive in both distance computations and I/Os. For the journal version we plan to add experimental results over larger real datasets and measure the evolution of the search performance as a function of n.

The weakest point of our structure is its high cost for insertions in terms of distance computations (whereas its number of I/Os is outstanding). We plan to study ways to optimize our idea of using an in-memory index to lower the cost of insertions. A variant of this structure can also be used to discard clusters at query time, without comparing their centers against the query.

Another important remaining work is to handle deletions, which is likely to work well with a lazy deletion mechanism that reconstructs clusters when they reach a fraction of marked elements. Adapting the original construction algorithm for the LC as a bulk-loading mechanisms for the DLC seems promising as well.

References

1. E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.
2. E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.
3. P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. 23rd VLDB*, pages 426–435, 1997.
4. V. Dohnal, C. Gennaro, P. Savino, and P. Zezula. D-index: Distance searching index for metric data sets. *Multimedia Tools and Applications*, 21(1):9–33, 2003.
5. R. F. Santos Filho, A. J. M. Traina, C. Traina Jr., and C. Faloutsos. Similarity search without tears: The OMNI family of all-purpose access methods. In *Proc. 17th ICDE*, pages 623–630, 2001.
6. M. Hetland. The basic principles of metric indexing. In *Swarm Intelligence for Multi-objective Problems in Data Mining*, volume 242 of *Studies in Computational Intelligence*, pages 199–232. Springer, 2009.
7. G. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580, 2003.
8. H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. iDistance: An adaptive B+-tree based indexing method for nearest neighbor search. *ACM Transactions on Database Systems*, 30(2):364–397, 2005.
9. M. Mamede. Recursive lists of clusters: A dynamic data structure for range queries in metric spaces. In *Proc. 20th Intl. Symp. on Computer and Information Sciences (ISCIS'05)*, LNCS 3733, pages 843–853, 2005.
10. G. Navarro and N. Reyes. Dynamic spatial approximation trees. *ACM Journal of Experimental Algorithmics*, 12:article 1.5, 2009.
11. G. Navarro and N. Reyes. Dynamic spatial approximation trees for massive data. In *Proc. 2nd SISAP*, pages 81–88, 2009.
12. G. Navarro and R. Uribe. Fully dynamic metric access methods based on hyperplane partitioning. *Information Systems*, 36(4):734–747, 2011.
13. G. Ruiz, F. Santoyo, E. Chávez, K. Figueroa, and E. Tellez. Extreme pivots for faster metric indexes. In *Proc. 6th SISAP*, pages 115–126, 2013.
14. H. Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann Publishers Inc., 2005.
15. T. Skopal, J. Pokorný, and V. Snásel. PM-tree: Pivoting metric tree for similarity search in multimedia databases. In *ADBIS (Local Proceedings)*, 2004.
16. P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems.* Springer, 2006.