New Dynamic Metric Indices for Secondary Memory [☆]

Gonzalo Navarro[a,1], Nora Reyes[b]

[a] *Center of Biotechnology and Bioengineering,*
Department of Computer Science, University of Chile, Chile
[b] *Departamento de Informática, Universidad Nacional de San Luis, Argentina*

Abstract

Metric indices support efficient similarity searches in metric spaces. This problem is central to many applications, including multimedia databases and repositories handling complex objects. Most metric indices are designed for main memory, and also most of them are static, that is, do not support insertions and deletions of objects. In this article we introduce new metric indices for secondary memory that support updates, that is, they are dynamic. First, we show how the dynamic and memory-based *Dynamic Spatial Approximation Tree (DSAT)* can be extended to operate on secondary memory. Second, we design a dynamic and secondary-memory-based version of the static *List of Clusters (LC)*, which performs well on high-dimensional spaces. The new structure is called *Dynamic LC (DLC)*. Finally, we combine the DLC with the in-memory version of DSAT to create a third structure, *Dynamic Set of Clusters (DSC)*, which improves upon the other two in various cases. We compare the new structures with the state of the art, showing that they are competitive and outstand in several scenarios, especially on spaces of medium and high dimensionality.

1. Introduction

The metric space approach has become popular in recent years to handle the various emerging databases of complex objects, which can only be meaningfully searched for by similarity [3, 21, 23, 9]. A large number of *metric indices*, that is, data structures to speed up similarity searches, have flourished. Most of them, however, are *static* solutions that work in main memory. Static indices have to be rebuilt from scratch when the set of indexed objects undergoes insertions or deletions. On the other hand, in-memory indices can handle only small datasets, suffering serious performance degradations when the objects reside on

[☆] Preliminary partial versions of this article appeared in *Proc. SISAP 2009* [17] and *Proc. SISAP 2014* [18].

Email addresses: gnavarro@dcc.uchile.cl (Gonzalo Navarro), nreyes@unsl.edu.ar (Nora Reyes)

[1] Funded with Basal Funds FB0001, Conicyt, Chile.

disk. Most real-life database applications require indices able to work on disk and to support insertions and deletions of objects interleaved with the queries.

The few metric indices supporting dynamism (that is, updates to the indexed set of objects) and designed for secondary memory can be classified into those using so-called pivots [8, 12, 20], those using hierarchical clustering [4, 19, 13], and those using combinations [5, 22].

Metric space searching becomes intrinsically more difficult as the so-called dimensionality of the space increases. Briefly, spaces are higher-dimensional when their histogram of distances is more concentrated. Low-dimensional spaces are easily dealt with by pivot-based indices, which use the distances to a small set of reference objects to map each other object (and the queries) into a coordinate space, where the problem is much better studied. On higher-dimensional spaces, however, the pivot-based approach fails, because more pivots are needed and thus the coordinate space is higher-dimensional as well. High-dimensional spaces, either metric or coordinate-based, are harder to index.

We are interested in dynamic secondary-memory indices for medium- and high-dimensional spaces. The few existing alternatives for this scenario are clustering-based indices. The best known metric index from this family is the *M-tree* [4]. Several others followed, for example the *EGNAT* [19], the *D-index* [5], the *PM-tree* [22], and the *MX-tree* [13]. Particularly, the *PM-tree* and the *MX-tree* are built on top of the *M-tree*.

In this article, we propose three new clustering-based dynamic indices for secondary memory. Our first index builds on the *Dynamic Spatial Approximation Tree* (*DSAT*) [16], a dynamic in-memory index that yields an attractive tradeoff between memory usage, update time, and search performance for medium-dimensional spaces. We design a secondary-memory variant, *DSAT+*, that retains the good performance of *DSAT* in terms of distance evaluations, while incurring a moderate number of I/O operations. This structure only supports insertions, which is acceptable in some scenarios.

Our second index builds on a simple static structure that performs very well on high-dimensional spaces, the *List of Clusters (LC)* [2]. A dynamic version, named *Recursive List of Clusters (RLC)* [14], exists, but it is also designed to work in main memory. We design a dynamic version of the *LC* that works in secondary memory, which we call *Dynamic List of Clusters (DLC)*. This structure requires a very low number of I/Os, but for large databases it incurs a significant number of distance computations.

Finally, our third index, called *Dynamic Set of Clusters (DSC)*, aims to obtain the best of both worlds. It combines a *DLC* with an in-memory *DSAT* in order to reduce the number of distance computations required by the *DLC*. The result is a structure with more balanced performance in searches and updates.

Our experimental comparisons show that our structures achieve reasonable disk page utilization and are competitive with state-of-the-art alternatives. For example, our *DSAT+* structure is the fastest to build and requires the least distance computations in selective queries, outperforming the *M-tree* [4], the best known alternative structure, in all aspects except the number of I/Os in some searches. Our *DSC* structure is more efficient at less selective queries and

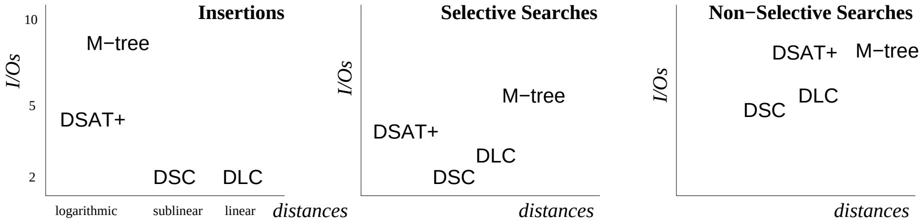


Figure 1: A rough graphical comparison of the insertion and search costs of the different structures.

in I/Os for construction, where it also outperforms the *M-tree*, but it requires more distance computations for construction. Figure 1 shows a rough graphical comparison of the insertion and search performance of our different structures, considering both distance computations and I/O operations.

We show that these conclusions carry over larger spaces and more recent competing data structures.

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. Well-known techniques [11, 10, 21] derive nearest-neighbor search algorithms from range search algorithms in a *range-optimal* way: the search cost is exactly that of range searching with a radius that captures the k nearest neighbors.

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. 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 N/B 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 threshold. In this article we address the more challenging case, where deletions must be physically executed. This is the case when deletions are frequent, or when the objects are large and retaining them for the sole purpose of indexing is unacceptable.

2.1. Experimental Setup

For the empirical evaluation of the indices we first consider three widely different metric spaces from the SISAP Metric Library (`www.sisap.org`) [7]. These are not very large, but are useful to tune the indices and make some design decisions based on their rough performance. Larger spaces are considered in Section 7.

Feature vectors, a set of 40,700 20-dimensional feature vectors, generated from images downloaded from NASA[2]. The Euclidean distance is used.

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. 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[3]. Any quadratic form can be used as a distance; we chose 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.

We consider range queries retrieving on average 0.01%, 0.1% and 1% of the dataset. This corresponds to radii 0.120, 0.285 and 0.530 for the feature vectors; and 0.051768, 0.082514 and 0.131163 for the color 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. As said, given the existence of range-optimal algorithms for k-nearest neighbor searching [11, 10], we have not considered these search experiments separately.

The disk page size is B = 4KB in most cases; we will also consider 8KB in some experiments. All the tree data structures cache the tree root in main memory. All the indices are built by successive insertions.

[2] At `http://www.dimacs.rutgers.edu/Challenges/Sixth/software.html`

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

As a baseline for comparisons, we use the *M-tree* [4]. This is the best-known dynamic secondary-memory data structure, and its code is freely available[4]. We use the parameter setting suggested by the authors[5]. We will also compare with the *eGNAT* [19] and the *MX-tree* [13].

3. Dynamic Spatial Approximation Trees

In this section we recall the key aspects of the *DSAT* [16], as we build on it for our secondary-memory indices.

The *DSAT* is a tree with one node per object. Searches start at the tree root and reach any node that is at distance at most r from the query q. The *DSAT* is based on the idea of approaching the query spatially, that is, moving towards the children of the current node that get closer to q, or closer to any potential element at distance r from q. Thus, the search backtracks over all the promising branches of the tree.

Insertions. A number of alternatives for insertion of new elements into a *DSAT* have been discussed and evaluated [16]. In this section we describe only the best one, which we have used in this paper. This is a combination of *timestamping* and *bounded arity.* A maximum tree arity (children per node) $MaxArity$ is fixed, and also a timestamp of the insertion time of each element is kept. Each tree node a maintains its neighbor set $N(a)$ (i.e., its children), its timestamp $T(a)$ (time a was inserted), and its covering radius $R(a)$ (maximum distance to a subtree element, used to prune the searches). Whenever a new element x is to be inserted in the subtree of a, we check whether it is closer to a than to any element in $N(a)$. If it is, we let x become a new element of $N(a)$ only if $|N(a)| < MaxArity$, in which case it is inserted at the end of $N(a)$ and its insertion time $T(x)$ is recorded. In any other case, x is recursively inserted into the subtree of its closest element in $N(a)$. Note that each element is older than its children and than its next sibling.

Figure 2 illustrates the construction of the *DSAT* on an example metric space in $\mathbb{R}^2$. Assume the elements arrive in order $p_1, p_2, p_3, \ldots, p_{15}$ and $MaxArity = 2$ or $MaxArity = 6$. The resulting trees are shown in Figure 3 and 4, respectively. Algorithm 1 details the insertion process.

The general idea is that elements are inserted into the subtree of their closest element in $N(a)$, or as a new element in $N(a)$ if they are closest to a. Thus the search looks for the closest element to the query q in $N(a)$, yet with a tolerance given by the search radius r. However, when a new element is added to $N(a)$, elements that were inserted in the subtrees of other neighbors in $N(a)$ might now prefer the new neighbor. The timestamp mechanism is used to avoid any rebuilding, for which the search mechanism is modified as detailed next.

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

[5] `SPLIT_FUNCTION = G_HYPERPL, PROMOTE_PART_FUNCTION = MIN_RAD, SECONDARY_PART_FUNCTION = MIN_RAD, RADIUS_FUNCTION = LB, MIN_UTIL = 0.2.`

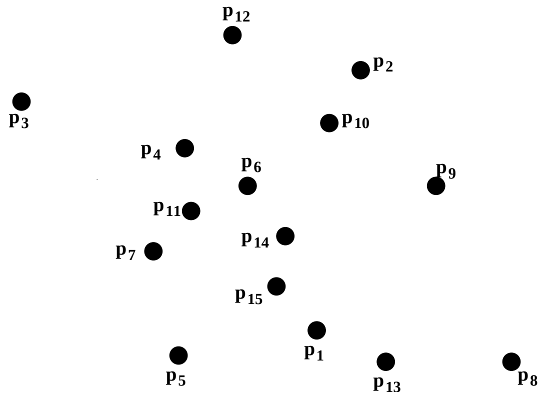


Figure 2: Example of a metric database in $\mathbb{R}^2$.

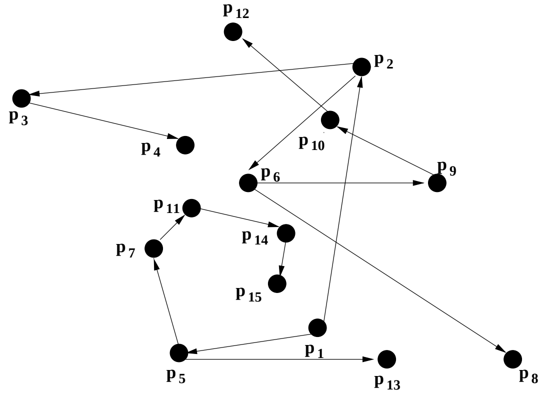


Figure 3: Example of the *DSAT* obtained with maximum arity 2.

Range searches. In principle, when searching for q with radius r, one should report the root a if $d(a,q) \leq r$, then find the closest element $b \in \{a\} \cup N(a)$, i.e., $b = \text{argmin}_{b \in a \cup N(a)} d(q,b)$, and enter every child c such that $d(q,c) \leq d(q,b) + 2r$ and $d(q,c) \leq r + R(c)$. Yet, because of the timestamped insertion process, we have to consider the neighbors $\langle b_1, \ldots, b_k \rangle$ of a from oldest to newest, and perform the minimization (to find b) while we traverse the neighbors, so as to decide at each point whether to enter into b_i or not. This is because, between the insertion of b_i and b_{i+j}, there may have appeared new elements that preferred to be inserted into b_i just because b_{i+j} was not yet a neighbor, so we may miss an element if we do not enter b_i because of the existence of b_{i+j}. Moreover, we use the timestamps to reduce the work done inside older neighbors at search time: Say that $d(q,b_i) > d(q,b_{i+j}) + 2r$. We have to enter b_i because it is older. However, only the elements with timestamp smaller than that of b_{i+j} should be considered when searching inside b_i; younger elements have seen b_{i+j} and they cannot be interesting for the search if they chose b_i. As parent nodes are older than their descendants, as soon as we find a node inside the subtree of b_i

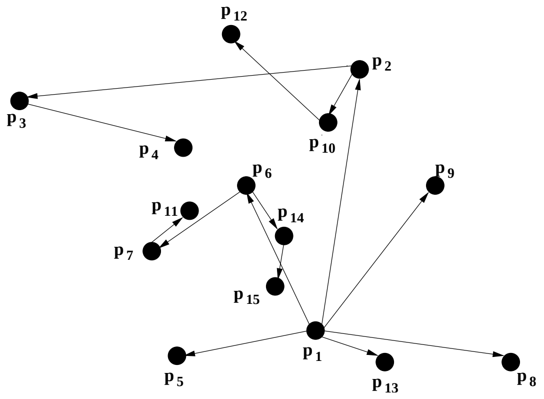


Figure 4: Example of the *DSAT* obtained with maximum arity 6.

Algorithm 1 Insertion of a new element x into a *DSAT* with root a using timestamping plus bounded arity.

```
Insert(Node $a$, Element $x$)

If $d(a,x) < d(c,x) \wedge |N(a)| < MaxArity$ Then
```

```
Insert(Node $a$, Element $x$)

$R(a) \leftarrow \max(R(a), d(a,x))$
$c \leftarrow \mathrm{argmin}_{b \in N(a)} d(b,x)$
If $d(a,x) < d(c,x) \wedge |N(a)| < MaxArity$ Then
    $N(a) \leftarrow N(a) : x$
    $N(x) \leftarrow \langle\rangle$, $R(x) \leftarrow 0$
    $T(x) \leftarrow CurrentTime$
    $CurrentTime \leftarrow CurrentTime + 1$
Else Insert($c$,$x$)
```

with timestamp larger than that of b_{i+j} we can stop the search in that branch, because its subtree is even younger.

Finally, as we bound the maximum arity, the root a is not included in the minimization, as an element may have been inserted into a child even if it should have become a neighbor. Algorithm 2 gives the search process. Note that $d(a, q)$ is always known except in the first invocation, and the initial t is $+\infty$.

Nearest-neighbor searches. The *DSAT* requires some care for k-nearest neighbor searching. In general, we perform a range search where the search radius r is the distance to the kth nearest element we have seen up to now, so r decreases along the process. The branches of the tree are visited in a particular order. We have a priority queue of nodes yet to traverse, and choose the next according to the minimum lower-bound distance to q. When this lower bound is larger than r, the search terminates.

However, in the *DSAT* this interacts with the timestamps. Instead of thinking in terms of maximum allowed timestamp of interest inside y, as done for range searches, now we think in terms of maximum search radius that permits

Algorithm 2 Searching for q with radius r in a *DSAT* rooted at a, built with timestamping plus bounded arity.

```
RangeSearch(Node $a$, Query $q$, Radius $r$, Timestamp $t$)

If $T(a) < t \ \wedge \ d(a,q) \leq R(a) + r$ Then
   If $d(a,q) \leq r$ Then Report $a$
   $d_{min} \leftarrow \infty$
   For $b_i \in N(a)$ Do // ascending timestamps
      If $d(b_i,q) \leq d_{min} + 2r$ Then
         $t' \leftarrow \min\{t\} \cup \{T(b_j), j > i \ \wedge \ d(b_i,q) > d(b_j,q) + 2r\}$
         RangeSearch($b_i$,$q$,$r$,$t'$)
         $d_{min} \leftarrow \min\{d_{min}, d(b_i,q)\}$
```

Algorithm 3 Searching for the k nearest neighbors of q in a *DSAT* rooted at a.

```
NNSearch(Node $a$, Query $q$, Neighbors wanted $k$)

$create(Q)$, $create(A)$
$insert(Q, (a, \max\{0, d(q,a) - R(a)\})$
$r \leftarrow \infty$
While $size(Q) > 0$ Do
   $(a, lbound) \leftarrow extractMin(Q)$
   If $lbound > r$ Then Break
   $insert(A, (a, d(q,a)))$
   If $size(A) > k$ Then $extractMax(A)$
   If $size(A) = k$ Then $r \leftarrow max(A)$
   $dmin \leftarrow \infty$
   For $b_i \in N(a)$ Do   // in increasing timestamp order
      $maxr \ \leftarrow \ \max \ \{(d(q,b_i) - d(q,b_j))/2, j > i\}$
      $insert(Q, (b_i, \max\{maxr, (d(q,b_i) - d_{min})/2, d(q,b_i) - R(b_i), t\})$
      $d_{min} \leftarrow \min\{d_{min}, d(q,b_i)\}$
Return $A$
```

entering y. Each time we enter a subtree y of b_i, we search for the siblings b_{i+j} of b_i that are older than y. Over this set, we compute the maximum radius that permits us not to enter y, namely $r_y = \max(d(q,b_i) - d(q,b_{i+j}))/2$. If it holds $r < r_y$, then we do not need to enter the subtree y.

Assume that we are currently processing node b_i and insert its children y into the priority queue. We compute r_y and associate it with y. Later, when the time to process y comes, we consider our current search radius r and discard y if $r < r_y$. If we insert a child z of y, then we associate it with the value $\max(r_y, r_z)$. Algorithm 3 shows the process. A is a priority queue of pairs *(node,distance)* sorted by decreasing *distance*. Q is a priority queue of pairs *(node,lbound)* sorted by increasing *lbound*.

Deletions. To delete an element x, the first step is to find it in the tree. Unlike most classical data structures for traditional search problems, doing this is not equivalent to simulating the insertion of x and seeing where it leads us to in the tree. The reason is that the tree was different at the time x was inserted. If x were inserted again, it could choose to enter a different path in the tree, which did not exist at the time of its first insertion.

An elegant solution to this problem is to perform a range search with radius zero. This is reasonably cheap and will lead us to all the places in the tree where x could have been inserted. On the other hand, whether this search is necessary is application-dependent. The application could return a handle when an object was inserted into the dataset. This handle can contain a pointer to the corresponding tree node. Adding pointers to the parent in the tree would permit us to locate the path for free (in terms of distance computations). Henceforth the location of the object is not considered as a part of the deletion problem, although we have shown how to proceed if necessary.

It should be clear that a tree leaf can always be removed without any cost or complication, so we focus on how to remove internal tree nodes. Note, however, that most tree nodes are leaves, especially when the arity is high.

Several alternatives to delete elements from a *DSAT* have been studied [16]. One of them aims at not degrading the searches. The best way to ensure that is to ensure that the tree resulting from the deletion of x is exactly as if x had never been inserted. This deletion method is called *rebuilding subtrees*: when node $x \in N(b)$ is deleted, we disconnect x from the main tree and reinsert all its descendants. Moreover, elements in the subtree of b that are younger than x have been compared against x to determine their insertion point. Therefore, these elements, in absence of x, could choose another path if we reinsert them into the tree. Thus, we retrieve all the elements younger than x that descend from b (that is, those whose timestamp is greater, which includes the descendants of x) and reinsert them into the tree, leaving the tree as if x had never been inserted.

If the elements younger than x are reinserted like fresh elements, that is, if they get new timestamps, then the appropriate reinsertion point beginning at the tree root must be found. On the other hand, if their original timestamp is maintained, then reinsertion can begin from b and save many comparisons. The reason is that they are reinserted as if the current time was that of their original insertion, when all the newer choices that appeared later did not exist, and hence those elements should make the same choice as at that moment, arriving again at b. In order to leave the resulting tree exactly as if x had never been inserted, the elements must be reinserted in the original order, that is, in increasing order of their timestamps.

Therefore, when node $x \in N(b)$ is deleted, all the elements younger than x are retrieved from the subtree rooted at b, disconnected from the tree, sorted in increasing order of timestamp, and reinserted one by one, searching for their reinsertion point from b. Algorithm 4 shows the deletion process.

There are two optimizations to this deletion method. Say that x will be deleted from the subtree rooted at node b (that is $x \in N(b)$). The first optimization makes a more clever use of timestamps. It can be observed that there are some elements younger than x that will not change their insertion point when we reinsert them into the subtree rooted at b. These elements are those

Algorithm 4 Algorithm to delete x from a *DSAT*, by rebuilding subtrees.

```
DeleteR(Node $x$)

1. $b \leftarrow parent(x)$
2. Collect in $S$ the elements of the subtree rooted at $b$, younger than $x$
3. Sort $S$ by increasing timestamps
4. $N(b) \leftarrow N(b) - \{x\}$
5. For $y \in S$ Do ReInsert($b$,$y$) // without changing its timestamp
```

older than the first child of x and also than the last sibling of x. For those elements, computing their new insertion point can be avoided (see [16] for more details). A second optimization is that the elements in $A(y)$ are closer to y than to any older neighbor, so y only needs to be compared against newer neighbors (as long as the same insertion path is repeated).

4. Dynamic Spatial Approximation Trees in Secondary Memory

Our secondary-memory variants of the *DSAT* maintain exactly the same structure and carry out the same distance evaluations of the main-memory version described in Section 3. The challenge is how to maintain a disk layout in order to minimize I/Os. We first describe the *DSAT** and, at the end, describe the variant called *DSAT+*. As said, we have not found efficient ways to handle deletions in the secondary-memory structures, so we consider only insertions and searches. In scenarios where deletions are not frequent, they can be handled by marking deleted elements so as to omit them from result sets of queries, and rebuilding the structures periodically.

We will force that, for any a, the set $N(a)$ will be packed together in a disk page, which ensures that the traversal of $N(a)$ requires just one disk read. Moreover, for technical reasons that become clear later, $MaxArity$ must be such that a disk page can store at least two $N()$ lists of maximum length. Yet, we are free to use a considerably lower value for $MaxArity$, which is usually beneficial for performance.

To avoid disk underutilization, we will allow several nodes to share a single disk page. We define insertion policies that maintain a partitioning of the tree into disk pages that is efficient for searching and does not waste much space. Indeed, we will guarantee a minimum average disk page utilization of 50%, and will achieve much more in practice.

4.1. Data Structure Layout

We represent the children of a node as a linked list. Therefore, each tree node has a first child $F(a)$ and a next sibling $S(a)$ pointers, where the latter is always local to the disk page. This allows making most changes to $N(a)$ without accessing a, which might be in another disk page. Each node also stores its timestamp $T(a)$ and its covering radius $R(a)$. Each disk page maintains the number of nodes actually used. Far pointers like $F(a)$ (i.e., that potentially

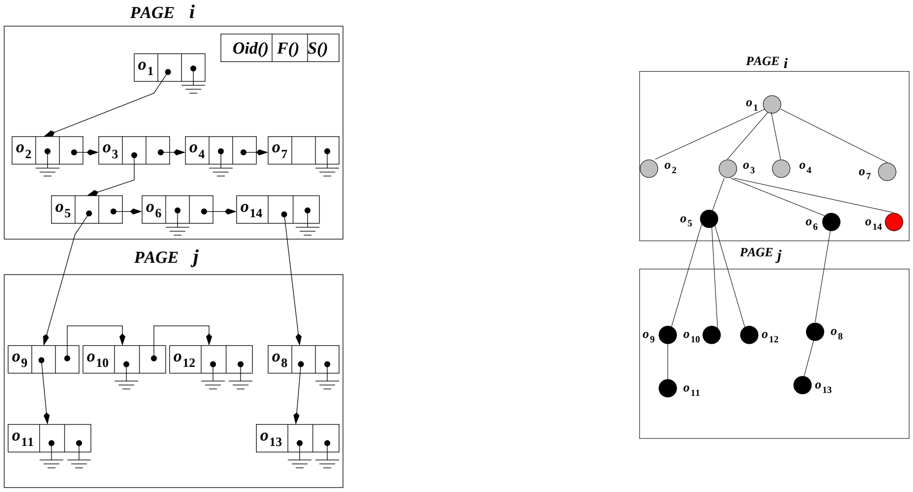


Figure 5: Example of $F(a)$ and $S(a)$ pointer layout.

point to another page) have two parts: the page and the node offset inside that page. Figure 5 illustrates the $F(a)$ and $S(a)$ pointers ($T(a)$ and $R(a)$ are omitted).

Nodes have fixed size in our implementation, thus varying-length objects like strings are padded to their maximum length. It is possible, with more programming effort, to allocate varying sizes for each node within a disk page. In this case the guarantee of holding at least two $N()$ lists per page translates into a variable bound on the arity of each node, so that a node a is not permitted to acquire a new neighbor if the total size of its $N()$ list would surpass half the disk page. In the case, however, of large objects that would force very low arities (or simply not fit in half a disk page), one can use pointers to another disk area, as it is customary in other metric structures, and treat the pointers as the objects. In this case every distance calculation implies also at least one disk access.

4.2. Insertions

To insert a new element x into the *DSAT** we first proceed exactly as in Algorithm 1: We find the insertion point in the tree, following a unique path, so that when we determine that x should be added to $N(a)$, we have both the pages of a and $N(a)$ loaded in main memory (these pages can be the same or different). Now we have to add x at the end of list $N(a)$ inside a disk page. If $N(a)$ was empty, then x will become the first child of a, thus we modify $F(a)$ and insert x into the page of a. Otherwise, x must be added at the end of $N(a)$, in the page of $N(a)$.

In either case, we must add x to an existing page, and it is possible that there is not enough space in the page. When this is the case, the page must be

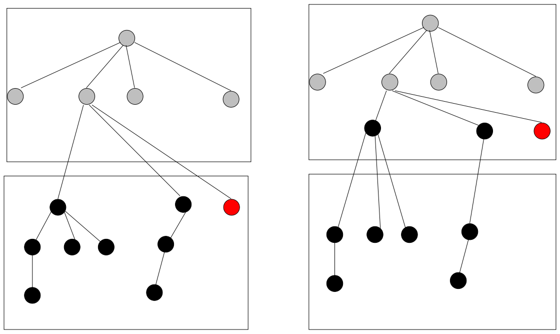


Figure 6: Example before (left) and after (right) applying the policy ***move to parent***.

split into two, or some parts of the page must be inserted into an existing page. We describe next our overflow management policy.

Because every $N(a)$ fits in a single disk page, the I/O cost of an insertion is at most h page reads plus 1-3 page writes, where h is the final depth of x in the tree. The reads can be much fewer than h since a and $N(a)$ can be in the same disk page for many nodes along the path.

4.3. Page Overflow Management

When the insertion of x in $N(a)$ produces a page overflow, we try out the following strategies, in order, until one succeeds in solving the overflow. In the following, assume x has already been inserted into $N(a)$, and now $N(a)$ does not fit in its disk page.

1st (move to parent) If a and $N(a)$ are located in different pages, and there is enough free space in the page of a to hold the whole $N(a)$, then we move $N(a)$ to the page of a and finish. This actually improves I/O access times for $N(a)$. We carry out 2 page writes in this case. See Figure 6.

2nd (vertical split) If the page of $N(a)$ contains subtrees with different parents from another page, we make room by moving the whole subtree where the insertion occurred to a new page (that is, we move all the subtree nodes residing in this page, to a new one). This maintains the number of disk reads needed to traverse the subtree. This needs up to 3 page writes, as the $F()$ pointer of the subtree parent resides in another page, and it must be updated. Note that we know where is the parent of the subtree to move, as we have just descended to $N(a)$. Figure 7 illustrates this case.

To maintain a property whose purpose will be apparent in Section 4.4, we avoid using the vertical split whenever the current page, after moving the chosen subtree to a new page, is less than half-full.

3rd (horizontal split) We move to a new page all the nodes of the subtree arrived at, with local depth larger than d, for the smallest d that leaves the current page at least half-full. The local depth is the depth within the subtree stored at the page, and can be computed on the fly at the moment of splitting.

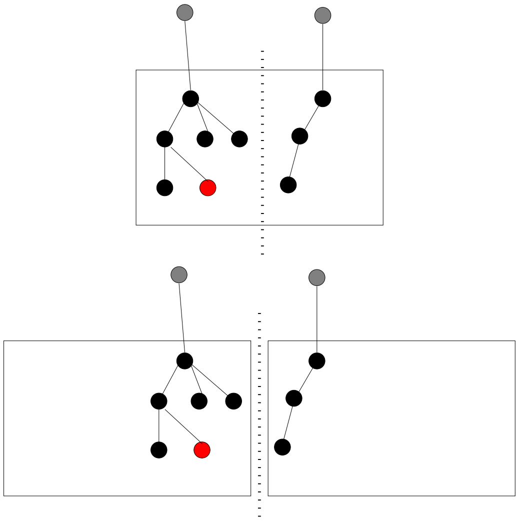


Figure 7: Example before (top) and after (bottom) applying the policy ***vertical split***.

Note that (i) the nodes whose parent is in another page have the smallest d and thus they are not moved (otherwise we would be moving the whole subtree, which is equivalent to a vertical split), hence only 2 page writes suffice; (ii) no $N(b)$ is split in this process because they have all the same d; (iii) the new page contains children of different nodes, and potentially of different pages after future splits of the current node.

We have to refine the rule when even the largest d leaves the current page less than half-full. In this case we can move only some of the $N(b)$ lists of depth d. This could still leave the current page underfull if there is only one $N(b)$ of maximum depth d, but this cannot happen because the disk page capacity is at least twice the maximum length of an $N()$ list. Another potential problem is that, if the maximum depth is $d = 0$, then we will move an $N(b)$ list whose parent is in another page. Yet, this is also impossible because the current subtree should be formed by only the top-level $N()$ list, and since it cannot account for more than half of the page, a vertical split should have applied in case of overflow.

Figure 8 illustrates a horizontal split.

Note that *move to parent* may apply because a *vertical* or *horizontal split* freed some space in a since its children had to move to a new page. A *vertical split* may apply after a *horizontal split* has put several nodes of different parents together. A *horizontal split* is always applicable because any individual $N(a)$ fits in a page. Finally, we try in the beginning to put $N(a)$ in the same page of a (when $N(a)$ is created) and later move it away via a *horizontal split* if necessary. Note that, in general terms, a *move to parent* improves I/O performance (as it puts subtrees together), a *vertical split* maintains it (as keeps subtrees together), and a *horizontal split* degrades it. Hence the order in which we try the policies.

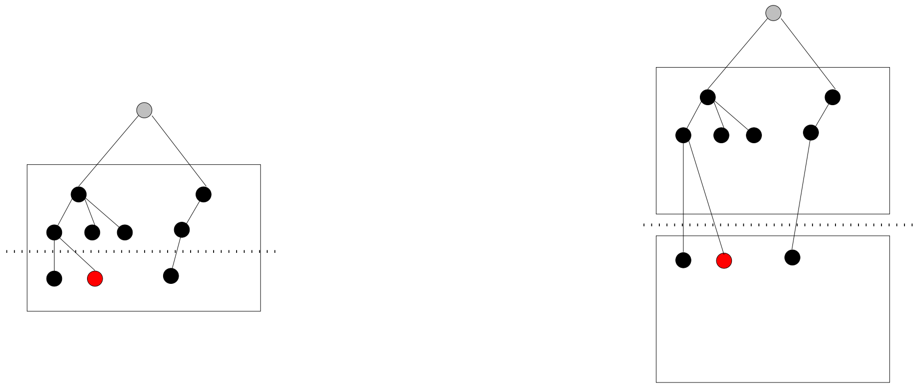


Figure 8: Example before (left) and after (right) applying the policy ***horizontal split***.

4.4. Ensuring 50% Fill Ratio

The previous operations do not yet ensure that disk pages are at least half-full. The *move to parent* case does: As in the child page $N(a)$ plus the rest overflowed, removing $N(a)$ cannot leave the child page less than half-full, as the page size is at least twice the size of $N(a)$. Yet, *vertical* and *horizontal* partitionings can create new underfull pages many times, although they do guarantee that the existing pages are always at least half-full.

To enforce the desired fill ratio, we will not allow indiscriminate creation of new pages. We will point all the time to *one* disk page, which will be *the only* one allowed to be less than half-full (this is initially the root page, of course). This will be called the *pointed* page, and we will always keep a copy in main memory to avoid rereading it, apart from maintaining it up to date on disk.

Whenever a new disk page is to be created, we try first to fit the data within the pointed page. If it fits, no page will be created. If it does not, we will create a new page for the new data, and it will become the pointed page if and only if it contains less data than the pointed page (thus only the pointed page can be less than half-full).

4.5. Searches

Searches proceed exactly as in the *DSAT*, for example, the range search is depicted in Algorithm 2. Let T be the rooted connected subgraph of the structure that is traversed during a search, and let L be the leaves of T (which are not necessarily leaves in the structure). Because of the disk layout of our structure, where sibling nodes are always in the same page, the number of page reads in the search is at most $1 + |T| - |L|$, and usually much less.

We assume we have sufficient space to store in main memory the disk pages containing the nodes from the current one towards the root, so that old disk pages must not be reread across the backtracking. This is not a problem, as the tree height is on average logarithmic at most [15].

4.6. The DSAT+ Variant

The *DSAT** we have described ensures 50% fill ratio, but this has a price in terms of compactness. Specifically, although our policies try to avoid it as much as possible, the tree may become fragmented especially due to the use of the pointed page mechanism. In this section we propose a heuristic variant, *DSAT+*, which tries to achieve better locality at the price of not ensuring 50% fill ratio (and indeed, as seen later, achieving lower fill ratios).

The differences with respect to the *DSAT** are as follows. In the *DSAT+* each subtree root at a page maintains a far pointer to its parent, and knows its global tree level. The vertical split applies every time when "move to parent" fails and there is more than one subtree root in the page. It divides the subtrees into two groups so that the partition is as even as possible in number of nodes, and creates a new page with one of the two groups. This page is a fresh one (no pointed page concept is used). In order to move arbitrary subtrees to another page we use their far parent pointers to update the child pointers of their parents. If there is only one subtree in the page, the horizontal split uses the global level to move all the nodes over some threshold to a new fresh page, again trying to make the sizes as even as possible.

4.7. Experimental Tuning

Figure 9 compares the search cost, in terms of both distance computations and disk pages read, of the *DSAT** and the *DSAT+*. We found empirically the best arity for each space, which turns out to be 4 for feature vectors and color histograms, and 32 for words, just as for the in-memory version of the structure [16]. As a baseline, we show the static in-memory *SAT* [15], which was already known to be outperformed by the in-memory dynamic *DSAT* [16].

In terms of disk pages read, the *DSAT+* outperforms the *DSAT** variant, as a consequence of improved locality and lower fragmentation. The difference is not very significant on color histograms, but is noticeable on the other spaces.[6]

Figure 10 compares construction costs (the static *SAT* builds as a whole, whereas the others build by successive insertions). It can be seen that both *DSAT* variants build fast (in two spaces, faster than the static *SAT* offline construction). Each insertion requires a few tens of distance computations and a few I/Os, and the cost grows very slowly as the database grows (proportionally to the tree depth). This time, the *DSAT+* variant is costlier in terms of I/Os than the *DSAT**, as it does not modify pointers in several disk pages. The difference, however, is generally small, being largest in the space of words.

Finally, the bottom of the figure shows the average disk page occupancy achieved for the different spaces. As explained, this is guaranteed to be at least 1/2 for the *DSAT**, but in practice it is 3/4 to 5/6. That of the *DSAT+* is around 2/3, which coincides with typical B-tree disk page occupancies ($1/\ln 2 \approx 69\%$).

[6] Note that the *DSAT** may read many more pages than the total amount. This is because subtrees share pages and thus the same page may be read several times along the process.

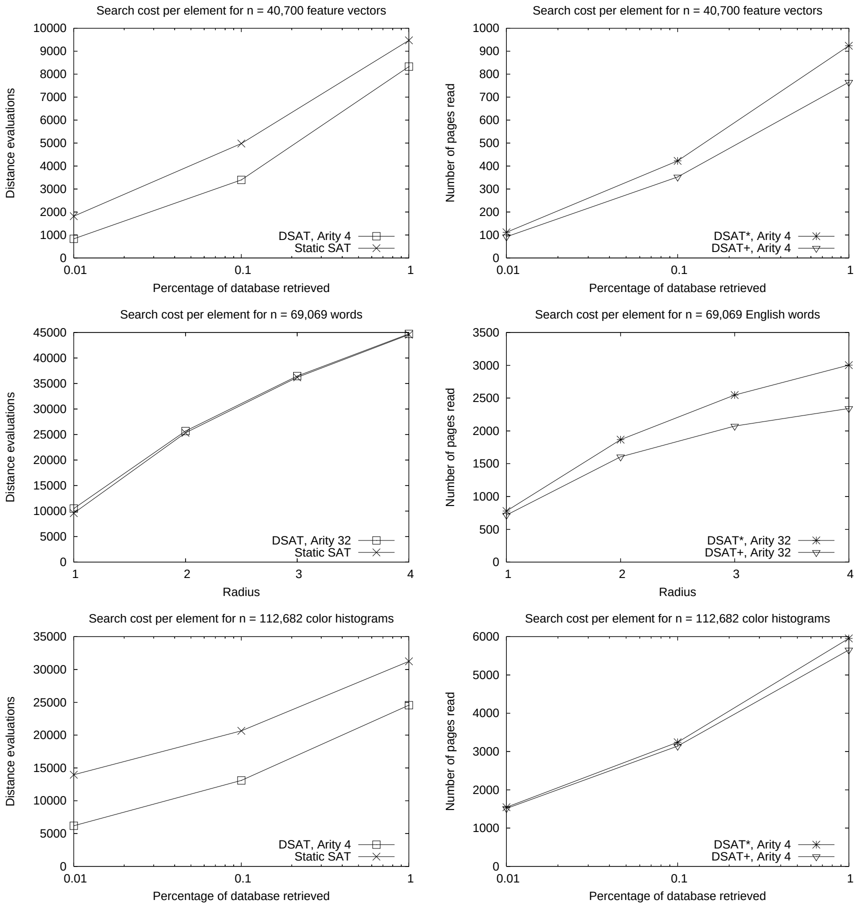


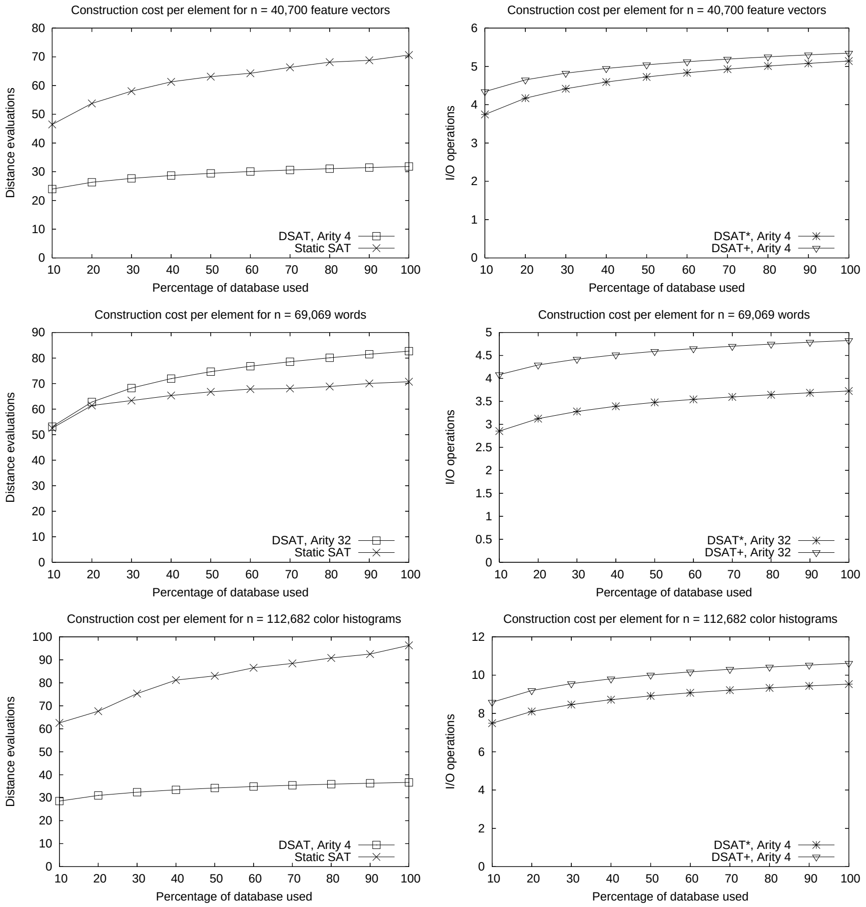
Figure 9: Search cost of *DSAT* variants in secondary memory, in terms of distance evaluations (left) and disk pages read (right). Both variants are identical in terms of distance evaluations, so the legend DSAT refers to both. We show one space per row.

We also show the total number of disk pages used. The *DSAT** uses significantly less space than the *DSAT+*.

For the comparison with other structures in Section 7, we will use the *DSAT+* with the best arity for each space, as it performs better for searches in exchange for being only slightly slower to build.

5. Dynamic List of Clusters in Secondary Memory

We introduce a dynamic and secondary-memory variant of the *List of Clusters* (*LC*), which aims at higher-dimensional spaces. Our secondary-memory version, *DLC*, retains the good properties of the *LC*, and in addition requires few I/Os operations for insertions and searches.



Dataset	Fill ratio		Total pages used	
	*DSAT**	*DSAT+*	*DSAT**	*DSAT+*
Vectors	80%	67%	1,271	1,726
Words	83%	66%	904	1,536
Colors	75%	67%	18,781	21,136

Figure 10: Construction cost of *DSAT* variants in secondary memory, in terms of distance evaluations (left) and I/Os (right). Both variants are identical in terms of distance evaluations, so the legend DSAT refers to both. We show one space per row. On the bottom, space usage of the secondary-memory structures.

5.1. List of Clusters

We briefly recall the *LC* structure [2]. It 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 (and not 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, $E = S - I$. Many alternatives to select centers and radii are considered [2], 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 $d(q,c) - r_c \leq r$ (that is, of the query ball and the cluster area intersect), we search exhaustively the set of internal elements I. The rest of the list is processed only if $r_c \leq d(q,c) + r$ (that is, if the query ball is not contained in the cluster area).

5.2. Dynamism and Secondary Memory

The *DLC* is based on the *LC* and also uses some ideas from the *M-tree* [4]. The challenge is to maintain a disk layout that minimizes both distance computations and I/Os, and achieves a good disk page utilization.

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.[7]

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, we will maintain components (1), (2) and (3) in main memory, that is, one object and a few numbers per cluster. Thus, we can determine whether a zone has to be scanned without reading data from disk. 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 (that is, we avoid reading a cluster if it does not intersect with the query ball, but we cannot stop the search when the query ball is contained in the cluster area).

[7] 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.

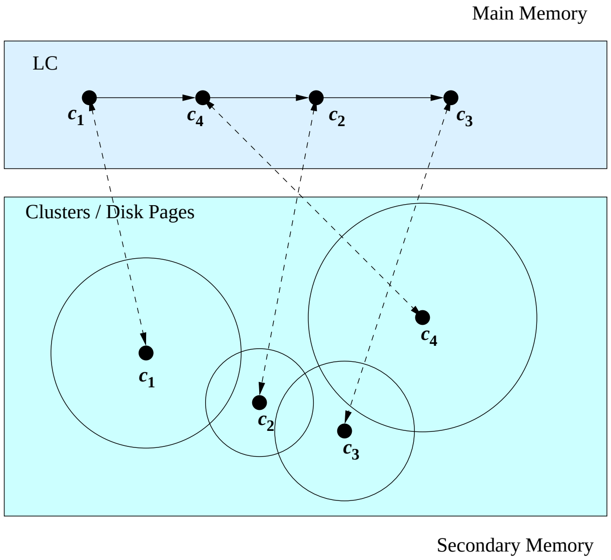


Figure 11: Example of a DLC in $\mathbb{R}^2$.

Figure 11 illustrates a DLC, with the set of clusters in secondary memory and the centers in main memory. Each cluster occupies at most a disk page. Figure 12 shows more details on the list of centers, where each element stores the cluster center $c = center(C)$, its covering radius $cr(C)$, its number of elements $\#(C)$, and the position of the disk page where the cluster C is stored on disk.

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.

5.3. 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 $center(C)$ 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.

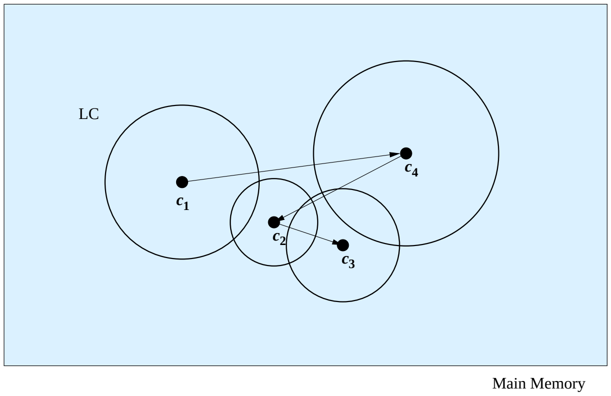


Figure 12: Details of the data stored in main memory for Figure 11.

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. Figure 13 illustrates an example in $\mathbb{R}^2$.

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 maintained in memory, and write C and N to disk. The combination of a selection and a partition method yields a *split*

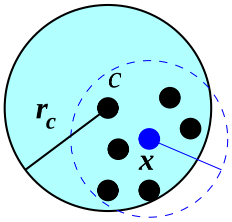


Figure 13: The possible shrink of a cluster zone due to an insertion of a new element x.

policy, several of which have been proposed for the *M-tree* [4].

5.4. Split Policies

The *M-tree* [4] 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 [4]) 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 [4], adding _1 or _2 to the strategy names depending on whether the partition strategy is confirmed or not.

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

SAMPL: 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 chosen, 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 element 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.

5.5. Searches

Upon a range 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 $d = d(q, \mathit{center}(C))$, and if $d \leq r$ we immediately report $c = \mathit{center}(C)$. Independently, if $d - \mathit{cr}(C) \leq r$, 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 $\mathit{cr}(C) > d + r$, as explained.

The scanning of a cluster also has a filtering stage: Since we store $d(x, c)$ for all $x \in \mathit{cluster}(C)$, 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.

Nearest-neighbor search algorithms can be systematically built over range searches in an optimal way [10]. 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 $d(q, \mathit{center}(C)) - \mathit{cr}(C)$, 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 $d(q, \mathit{center}(C)) - \mathit{cr}(C)$ value of an unexplored cluster.

5.6. Deletions

An element x that is deleted from the *DLC* may be a cluster center or an internal object of a cluster. If x is an internal object of a cluster C, we read its corresponding disk page, remove x, and write the page back to disk. In the in-memory structures, we update $\#(C)$ and recompute $\mathit{cr}(C)$ if necessary (that is, if $\mathit{cr}(C) = d(x, c)$, where $c = \mathit{center}(C)$). Note that we do not need to calculate any new distance to carry out this process.

If, instead, x is the center of a cluster C, we read its disk page and choose a new center for C among its elements. The new center will be the element $y \in \mathit{cluster}(C)$ that minimizes the maximum distance to the other elements in $\mathit{cluster}(C) - \{y\}$, that is the element whose cr will be minimum. Then, we write back the page to disk and update $\mathit{center}(C)$, $cr(C)$, and $\#(C)$ in main memory. In this case we need $O(m^2)$ distance calculations. As it can be noticed, in both cases we need only two I/O operations.

If we want to ensure a minimum fill ratio $\alpha \leq 0.5$ in the disk pages, we must also intervene when the deletion in a cluster leaves its disk page underutilized. In this case, we delete the whole cluster and reinsert its elements in the structure. This is not as bad as it might seem because the elements of the cluster tend to be close to each other, and thus it is usual that several elements are inserted in the same page, thus saving I/Os. In the worst case, however, $2\alpha m$ I/O operations and αn distance evaluations might be necessary. If we choose not to ensure any fill ratio, then a cluster C will be discarded only when it runs out of elements.

5.7. Experimental Tuning

Figure 14 shows search costs for a number of combinations of split policies. We only show the best ones to avoid a flooding of graphs. As a baselines, we show the in-memory static *LC* [2]. For the *LC*, we use as cluster size the maximum number of elements that fit in each cluster of *DLC*.

The compactness policy is always better than the occupancy policy when searching for the insertion point, so we only show the former. Similarly, the balanced partitioning obtains worse search costs than the others, because it prioritizes occupancy over compactness.

In general, the performance for searches changes only moderately from one policy to another. The static *LC*, instead, is significantly better in terms of distance computations, showing that the offline construction does help it find a better clustering of the objects than the say *DLC* handles the online insertions.

Figure 15 shows construction costs. We only show distance computations, since the I/O cost is always 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. This is basically the minimum possible cost for an individual insertion. In exchange, the number of distance computations used to insert an element is in the hundreds and even in the tens of thousands. The worst offenders are the non-confirmed strategies, especially in cases where the objects are small and thus the disk pages (i.e., the clusters) contain a large number of elements. Still, even the better *DLC* alternatives require hundreds of distance computations per insertion, an order of magnitude more than the *DSAT* variants.

The figure also shows the average disk page occupancy achieved, considering strategy SAMP_1 HYPERPL for feature vectors, SAMP_1 HYPERPL for words, and mM_RAD_2 MIXED for color histograms. Note that the fill rations of the *DLC* are not as high as for the *DSAT+*. However, each object stores less data in the *DLC* than on the *DSAT+*. As a consequence, the *DLC* uses less total space

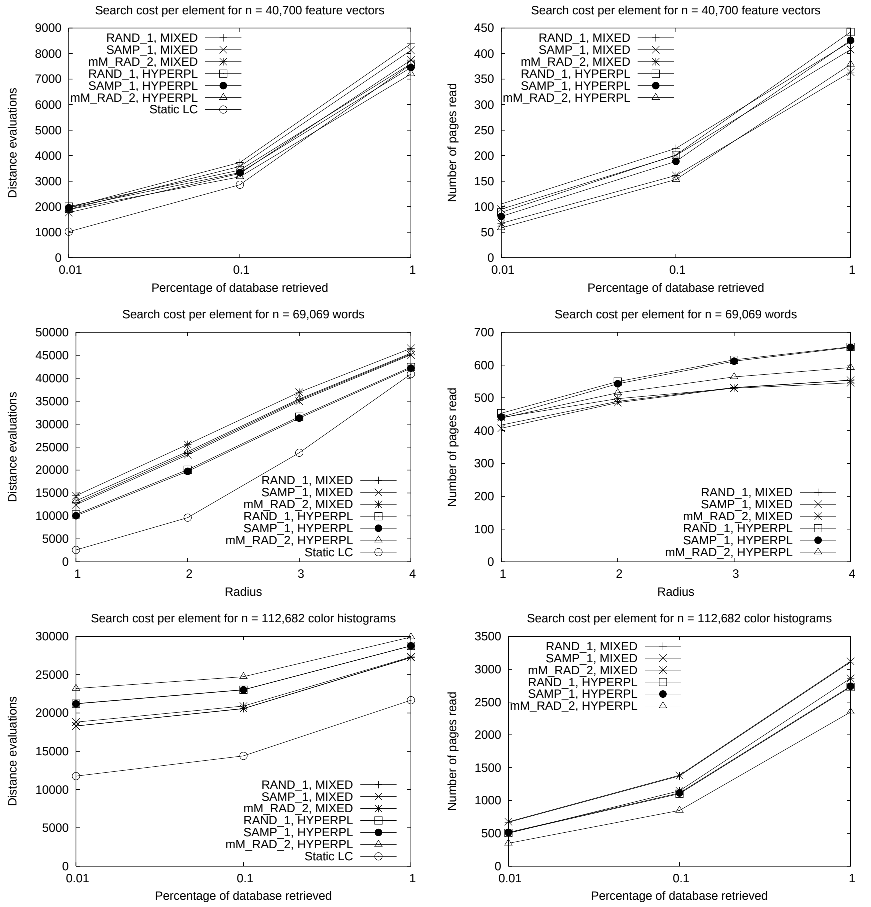


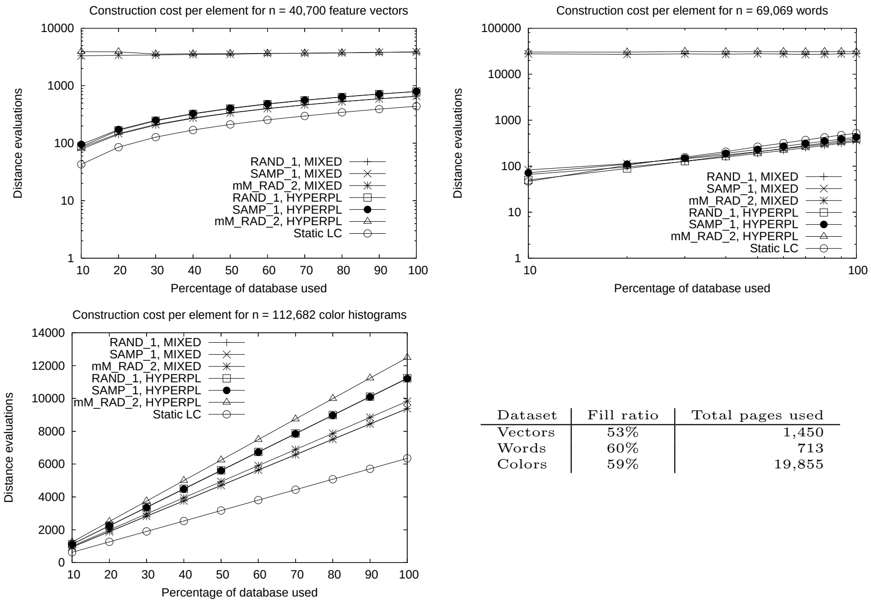
Figure 14: Search cost of *DLC* variants in secondary memory, in terms of distance evaluations (left) and disk pages read (right). We show one space per row.

than the *DSAT+*. We remind that, by using the MIXED partition strategy, we can guarantee a minumum disk page occupancy, if desired.

It is not easy to choose a small set of variants for the comparisons of Section 7. We choose two for each space, which have a reasonably good performance in all cases: SAMP_1 HYPERPL and SAMP_1 MIXED for the feature vectors; SAMP_1 MIXED and SAMP_1 HYPERPL for words; and mM_RAD_2 MIXED and SAMP_1 HYPERPL for color histograms.

6. Dynamic Set of Clusters in Secondary Memory

As the tuning experiments have shown, the secondary-memory variants of the *DSAT* we have presented in Section 4 perform well for insertions and searches in terms of distance computations, but the number of I/Os incurred



Dataset	Fill ratio	Total pages used
Vectors	53%	1,450
Words	60%	713
Colors	59%	19,855

Figure 15: Construction cost of *DLC* variants in secondary memory, in terms of distance evaluations. The table shows the space occupancy. Beware of logscales.

is high. On the other hand, the *DLC* of Section 5 uses a low number of I/O operations, but the number of distance computations for insertions is high. This is because we must compare the query with all the cluster centers c, of which there are $O(n/m) = O(n/B)$. While this happens both in queries and updates, queries are usually expensive and this penalty is not as noticeable as in updates.

Our idea is replace the sequential scan of the relevant centers by an efficient in-memory structure, both for queries and updates. As the centers will not be scanned sequentially, their order becomes immaterial, and the structure is called *Dynamic Set of Clusters (DSC)*. Our in-memory structure must support searches, but also insertions and deletions, for the case where the centers of the clusters change, or we insert or remove clusters. We have chosen the *DSAT* described in Section 3.

6.1. Data Structure Layout

The *DSC* uses the same disk structure of the *DLC*, and clusters will store the same in-memory data as well. Instead of arranging the clusters on a list, however, the set of all centers $c = center(C)$ will be maintained in an in-memory *DSAT* data structure. In addition to $center(C)$, $cr(C)$, and $\#(C)$, each cluster will store in main memory the value $CRZ(c)$, the maximum distance between c and any element stored in the cluster of any center in the subtree of c.

Figure 16 illustrates a *DSC*, showing the set of clusters in secondary memory and its *DSAT* in main memory. Each cluster occupies at most a disk page.

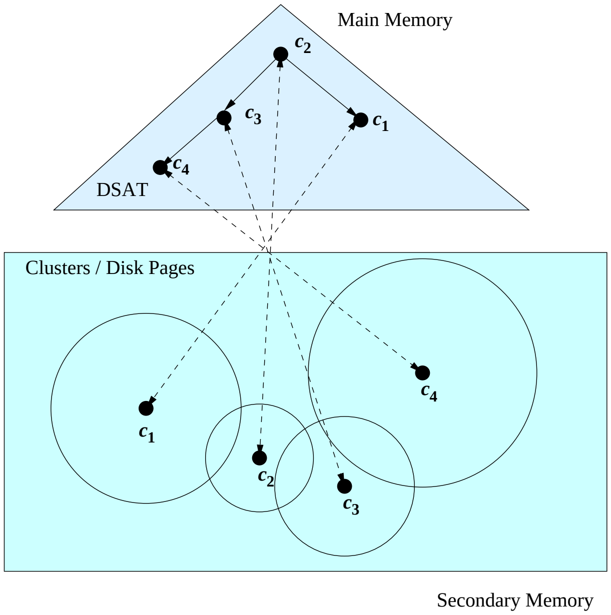


Figure 16: Example of a DSC in $\mathbb{R}^2$.

Figure 17 expands the details of the corresponding $DSAT$, where the values $CRZ(c) \neq 0$ are shown. Besides, we also show the value $CR(c) \neq 0$ of each c in the $DSAT$, where $CR(c)$ is the covering radius of c in the in-memory $DSAT$.

The structure 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. Besides, these are located in consecutive disk pages. Therefore, if any problem occurs and the $DSAT$ of centers is lost or corrupted, we can rebuild it from the clusters information in a sequential scan of clusters.

6.2. Insertions

The $DSAT$ structure supports our policy of inserting a new element x into the cluster with center c closest to x. Therefore, a $k = 1$ nearest neighbor query on the $DSAT$ (Algorithm 3) yields the center where x should be inserted. This incurs no I/Os, and requires much fewer distance computations than the sequential scan carried out in the DLC.

Once the cluster C that will receive the insertion is determined, we read the corresponding page from disk, add x in it, write it back to disk, and update the in-memory values $\#(C)$ and $cr(C)$, just as the DLC. However, we must also consider whether x would be a better center of C than $c = center(C)$. In such a case, we replace the cluster center by c, as explained in Section 5. However,

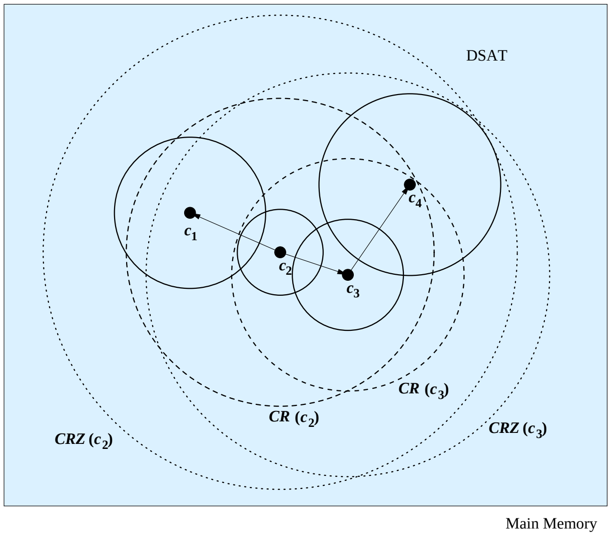


Figure 17: Details of the in-memory $DSAT$ of Figure 16.

in the case of the DSC, we must also update the $DSAT$ of centers, by deleting the old center c from the tree and inserting x as a new one. These updates on the $DSAT$ only require distance computations.

The insertion of a new element x in a cluster may increase $cr(C)$, and this triggers an update of the CRZ fields for all the ancestors of c in the tree. For each ancestor c', we set $CRZ(c') \leftarrow \max\{CRZ(c'), d(c', x)\}$. Note that all these distances $d(c', x)$ are already computed by the time the search reaches node c.

When the cluster chosen for insertion is full, the procedure is similar to the DLC. We split it into two clusters, the current one (C) and a new one (N), choose centers for both (according to a "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 "partition method"). Finally, write C and N to disk. To complete the procedure, we remove the old center c from the $DSAT$ and insert the new elements $center(C)$ and $center(N)$ (we may avoid removing c and inserting $center(C)$ if both are the same).

We use the Algorithms 1 and 4 to insert a new center into the $DSAT$ of centers and to delete an old center, respectively. We use also the optimizations described for deletion.

Note that, when we insert a new center in the $DSAT$ with its cluster already defined (as it occurs after a split), we could exactly recompute the CRZ fields of all the nodes in the insertion path, by comparing them with each element in $cluster(C)$. However, this is too expensive. Instead, each field is recomputed as $CRZ(c') \leftarrow \max\{CRZ(c'), d(c', c) + cr(c)\}$, which gives an upper bound. Similarly, when we delete elements in the $DSAT$, the fields CRZ are not easily updated. In this case we simply leave the old values, ignoring the fact that the new ones might be smaller. Recall that this $DSAT$ structure is volatile; when the system is used again, a fresh $DSAT$ is built from the centers that are stored on disk. Therefore, this inefficiency in updating the CRZ fields is not cumulative.

6.3. Searches

Searches for q with radius r can also take advantage of the $DSAT$ structure to find all the zones that intersect the query ball. The only difference is that each node c stands for a ball around c of radius $cr(c)$. Therefore, we can use essentially the same Algorithm 2, with the difference that we must use $CRZ(a)$ instead of $R(a)$ in line 1. Also, the range search collects any element a such that $d(q, a) \leq r + cr(a)$, in line 2.

As done with the DLC, we first collect all the centers c whose clusters must be scanned, sort them by their position on disk, and then read them one by one into main memory, in order to reduce disk seeks. The centers that satisfy $d(q, c) \leq r$ are also reported. As before, we use the distances $d(x, c)$ stored with the clusters on disk to try to avoid computing $d(x, q)$ for each $x \in cluster(C)$.

Nearest neighbor searches can also take advantage of the $DSAT$ structure. Instead of inserting all the centers c in the priority queue, we insert the center of the $DSAT$ root. The priority of the nodes c will be their lower-bound distance to q, $d(c, q) - CRZ(c)$. We extract the next element, exhaustively examine its corresponding cluster (using the filtering based on precomputed $d(x, c)$ information), and insert its $DSAT$ children into the queue.

6.4. Deletions

When an element x in the DSC is deleted, the process is very similar to that of the DLC, but the information in the $DSAT$ of centers is updated if necessary. If x is not the center of its cluster C, no action on the $DSAT$ is needed (again, we do not try to recompute the possibly smaller CRZ fields of all the ancestors of $center(C)$).

Instead, if $x = center(C)$, we choose a new center for C. In this case, we delete x in the $DSAT$ and insert the new $center(C)$, with its corresponding cr value. This incurs no further I/Os.

When a cluster has to be discarded and reinserted because its number of elements falls below a threshold, we try to avoid searching their insertion points one by one in the $DSAT$ of centers. We choose a central element x among those to be reinserted and find its nearest center c in the $DSAT$. Then, we insert x in $cluster(C)$, and as many closest elements to c as possible, among those to be reinserted. If $cluster(C)$ becomes full, we restart the process with the remaining elements. However, if the cluster splits right after inserting x, we exploit the fact that we have two close clusters less than half-full and insert all the remaining elements in the two new clusters. Each element is then inserted in the cluster of its closest center.

In the worst case, a deletion can force a complete rebuilding of the in-memory $DSAT$. If m is the maximum size of a cluster and the dataset contains n elements, the total reconstruction of this *DSAT* of centers requires $O((n/m)\log(n/m))$ distance computations. Instead, only 5 I/Os are required for a deletion in the worst case. These may be needed if we have to remove a cluster C because it contains too few element: (1) To keep the pages consecutively on disk, we have to read the last page and write it at the page of the deleted cluster; (2) we have to read the disk page of the cluster whose center is closest to $center(C)$; (3) when inserting the elements of C into this new cluster, it may overflow. Splitting it requires writing two pages on disk.

6.5. Experimental Tuning

As for the *DLC*, we compare the effect of the various policies for *DSC*, showing only those that performed best. Figure 18 shows the search costs in terms of distance evaluations and pages read. Both costs are similar to those of the *DLC*.

The best alternatives are, in general, mM_RAD_2, RAND_1, and SAMP_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.

Figure 19 shows the construction costs per element in terms of distance evaluations. As for the *DLC*, the number of I/Os is basically 1 read and 1 write per insertion. As it can be observed, the MIXED partition strategies obtain better costs during insertions and, as before, the center selection strategy of mM_RAD_2 is the most expensive.

While the construction costs are still in the hundreds, and thus an order of magnitude higher than for the *DSAT*, they are in general significantly lower than for the *DLC*. In particular, we note that the linear increases in insertion cost of the *DLC* has turned to clearly sublinear (although not as low as the apparently logarithmic cost of insertion in the *DSAT*).

The figure also shows a table with space occupancy, which is similar for *DLC* and *DSC*. Here we use the variants mM_RAD_2 HYPERPL for feature vectors, mM_RAD_2 MIXED for words, and mM RAD_2 MIXED for color histograms.

For the comparisons of Section 7, we choose the following variants, considering a balance of distance computations and I/Os: for feature vectors we select mM_RAD_2 MIXED and mM_RAD_2 HYPERPL (these had excessively high insertion cost in the *DLC*, but not in the *DSC*); for words we choose mM_RAD_2 HYPERPL and mM_RAD_2 MIXED; and for color histograms we select RAND_1 HYPERPL and mM_RAD_2 MIXED.

7. Experimental Comparison

In this section we empirically evaluate the best representatives of our indices, comparing them with previous work on the small spaces we used for tuning and in larger ones.

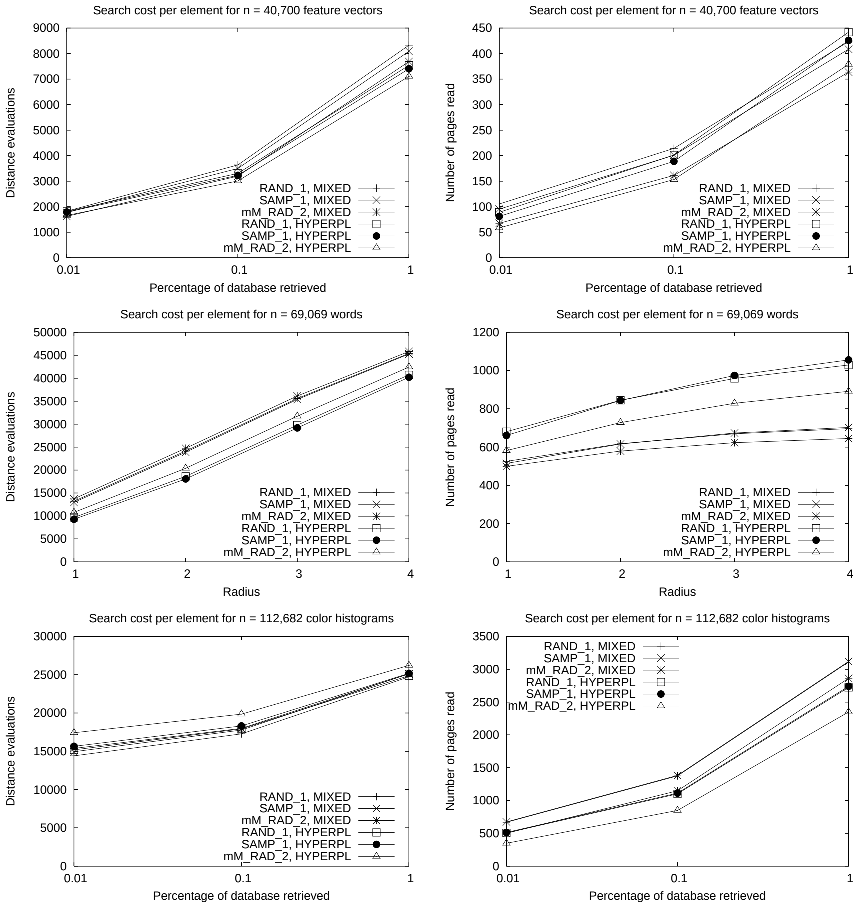


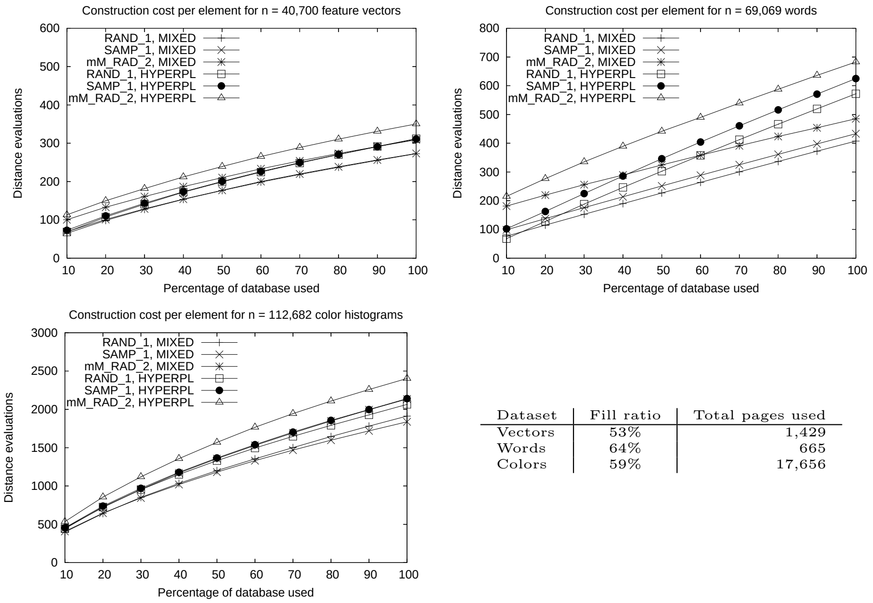
Figure 18: Search cost of *DSC* variants in secondary memory, in terms of distance evaluations (left) and disk pages read (right). We show one space per row.

7.1. Comparing with M-tree

We start with a comparison with the classical baseline of the literature, the *M-tree* [4], whose code is easily available.

Figure 20 compares the search costs. In terms of distance computations, the *DSAT+* always outperforms the *M-tree*. The *DSAT+* works better for the more selective queries. For example, on feature vectors, the *DSAT+* is the best for the most selective queries, but for larger radii the *DSC* takes over. In words, instead, the *DSAT+* is outperformed by the *DSC* even for radius $r = 1$, but in color histograms the *DSAT+* is the best even to retrieve 1% of the dataset. The *DLC* closely follows the *DSC*, and both outperform the *M-tree* in almost all cases.

When we consider disk pages read, both the *DSAT+* and the *M-tree* are outperformed by the *DLC* and *DSC* variants. These are barely distinguishable,



Dataset	Fill ratio	Total pages used
Vectors	53%	1,429
Words	64%	665
Colors	59%	17,656

Figure 19: Construction cost of *DSC* variants in secondary memory, in terms of distance evaluations. The table shows the space occupancy.

except on words, where the *DLC* is better. The comparison between the *DSAT+* and the *M-tree* is mixed. In color histograms, the *M-tree* is twice as slow, on words it is twice as fast, and in the feature vectors the result depends on the selectivity of the query.

Figure 21 compares construction costs, both in distance computations and I/Os. In terms of distance computations, the *DSAT+* is unbeaten: it requires a few tens of distance computations per insertion. It outperforms the *M-tree* approximately by a factor of 2. Instead, the *DSC* requires a few hundred distance computations, and usually outperforms the *DLC*, which is the most expensive (except on words, where it is faster than the *DSC*). It is also apparent that the cost growth in the *DSAT+* and the *M-tree* is much lower than on the *DSC*, and this is lower than on the *DLC*. It can be observed that, although the *M-tree* requires fewer distance evaluations to insert an element than the *DLC* and the *DSC*, it needs more I/Os because it uses more disk pages. This is because the *M-tree* is fast to determine an insertion point for the new element, but it does not find the best one. Instead, the *DSC* and the *DLC* have a more costly insertion because they spend more time to determine the best cluster to insert the new element. The experimental results in Figure 20 validate this conjecture, because searches are cheaper, both in I/O operations and distance computations, in the *DSC* and the *DLC* than in the *M-tree*.

In terms of I/Os, the situation is basically the opposite. Both the *DLC* and the *DSC* require almost 2 I/O operations per insertion, whereas the *DSAT+*

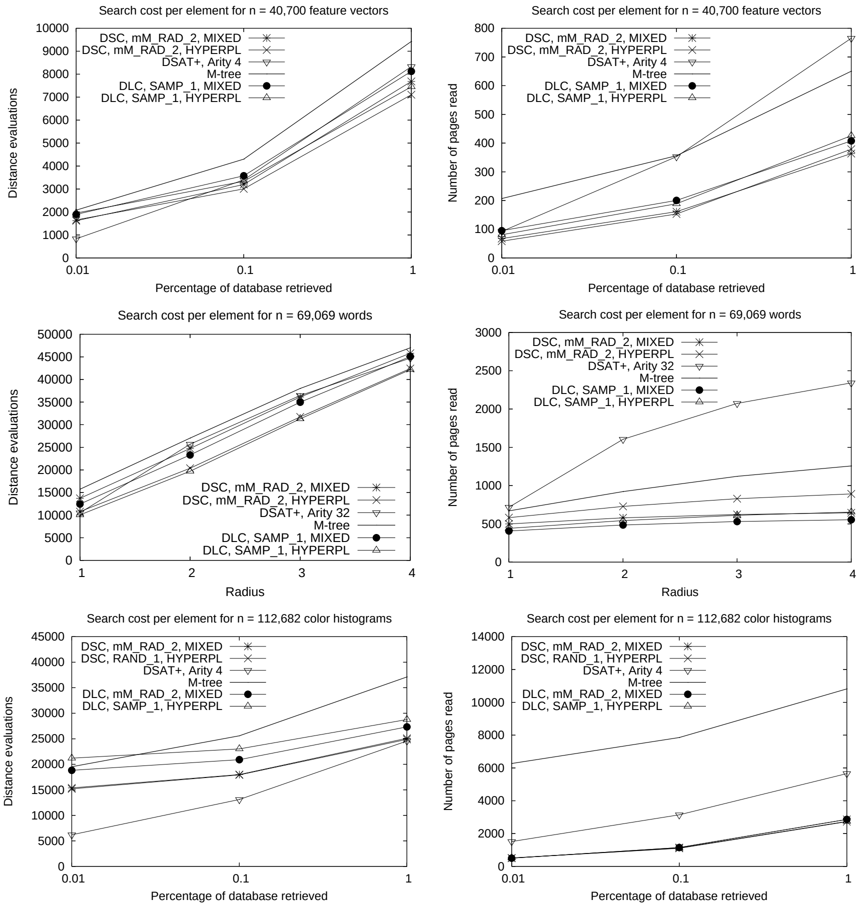


Figure 20: Comparison of search costs of *DSC*, *DLC*, *DSAT+*, and *M-tree*. We show distance computations on the left and disk page reads on the right.

requires 4 to 10, and the *M-tree* requires more, 6 to 20.

Table 1 shows the space usage of the indices, both in terms of fill ratio and total number of disk pages used. It can be seen that the *DSAT+* achieves the best disk page utilization, around 67%. This drops to 53%–64% for the *DLC* and the *DSC*. Interestingly, when we consider the total number of disk pages used, it turns out that the *DSAT+* is the most space-consuming structure, whereas the *DLC* and *DSC* use fewer pages. This is because the latter structures store less data per object. In this comparison, the *M-tree* is indeed the most space-consuming of the data structures.

Overall, we can draw the following conclusions:

1. The *DSAT+* is a structure offering a good balance between insertion and search costs, usually outperforming the *M-tree* (except on I/Os for searches, in some cases). It performs better on more selective queries. It

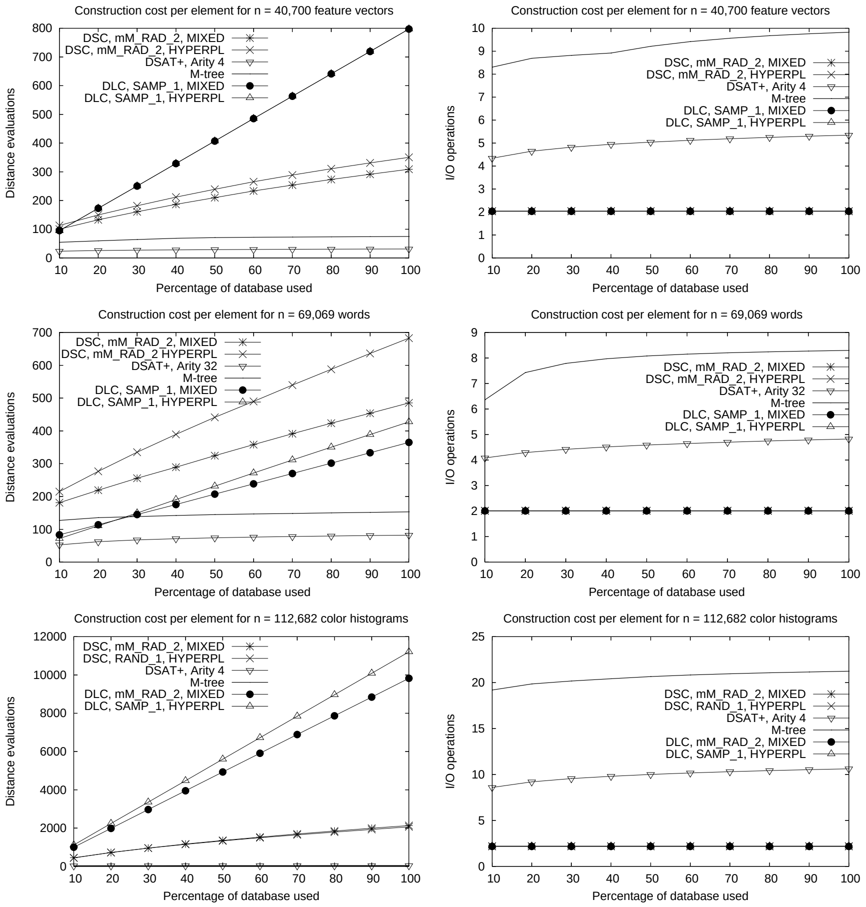


Figure 21: Construction costs of *DSC*, *DLC*, *DSAT+*, and *M-tree*. We show distance computations on the left and I/Os on the right.

is most competitive for distance computations, whereas in I/Os it is generally outperformed by the *DLC* and the *DSC*. Finally, the *DSAT+* lacks an efficient support for deletions.

2. The *DSC* performs better when the queries are less selective, but even when not, it usually outperforms the other structures at searches, both in distance computations and I/Os. It also outperforms them at I/Os for insertions, where it performs essentially two operations. Its weakest point is the number of distance computations required for insertions. Although it is sublinear, and generally better than those required by the *DLC*, the number is an order of magnitude higher than those required for the *DSAT+* and the *M-tree*.

Dataset	Fill ratio			Total pages used			
	DSAT+	*DLC*	*DSC*	*DSAT+*	*DLC*	*DSC*	*M-tree*
Vectors	67%	53%	53%	1,726	1,450	1,429	1,973
Words	66%	60%	64%	1,536	713	665	1,608
Colors	67%	59%	59%	21,136	19,855	17,656	31,791

Table 1: Space usage of the selected indices.

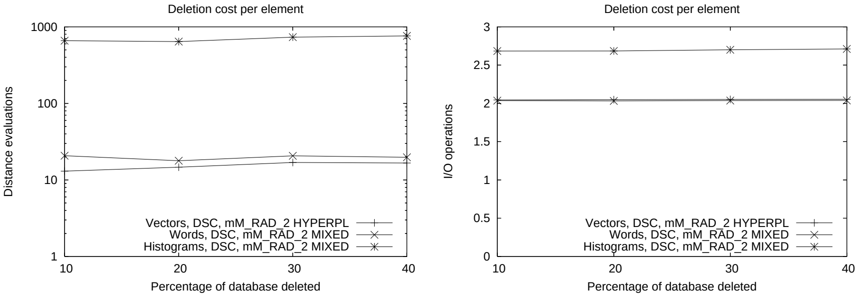


Figure 22: Deletion costs for the best alternatives of the *DSC*. We show distance computations on the left and I/Os on the right.

7.2. Deletions

Since the *DSAT+* does not support deletions, and the *DLC* is generally outperformed by the *DSC*, we study the effect of deletions on the latter structure. We study both the cost of a deletion and the degradation suffered by deletion and search costs after a significant number of deletions is performed. For this type of experiments, on an index of n elements, we randomly select a given fraction of the elements and delete them from the index, so that *after* the deletions the index contains n elements. For example, if we search half the space of strings after 30% of deletions, it means that we inserted 49,335 elements and then removed 14,800, so as to leave the same 34,534 elements (half of the set). The index where we search contains always the same half of space. For each dataset we use the alternative considered in the space tables: mM_RAD_2 HYPERPL for feature vectors, SAMP_1 HYPERPL for words, and mM RAD_2 MIXED for color histograms.

Figure 22 shows the deletion costs as the percentage of deletions increases. The number of I/Os required by deletions stays in 2–3, whereas the number of distance computations needed to restore the correctness of the index after removing the element is relatively low: a few tens for feature vectors and words, and a few hundreds (close to the cost of an insertion) for color histograms. These numbers do not change significantly when the percentage of deletions increases.

Figure 23 shows how massive deletions degrade range searches. The degradation is noticeable in some cases, both in distance computations and I/Os, reaching as much as 50% extra I/Os on feature vectors when we delete 40% of

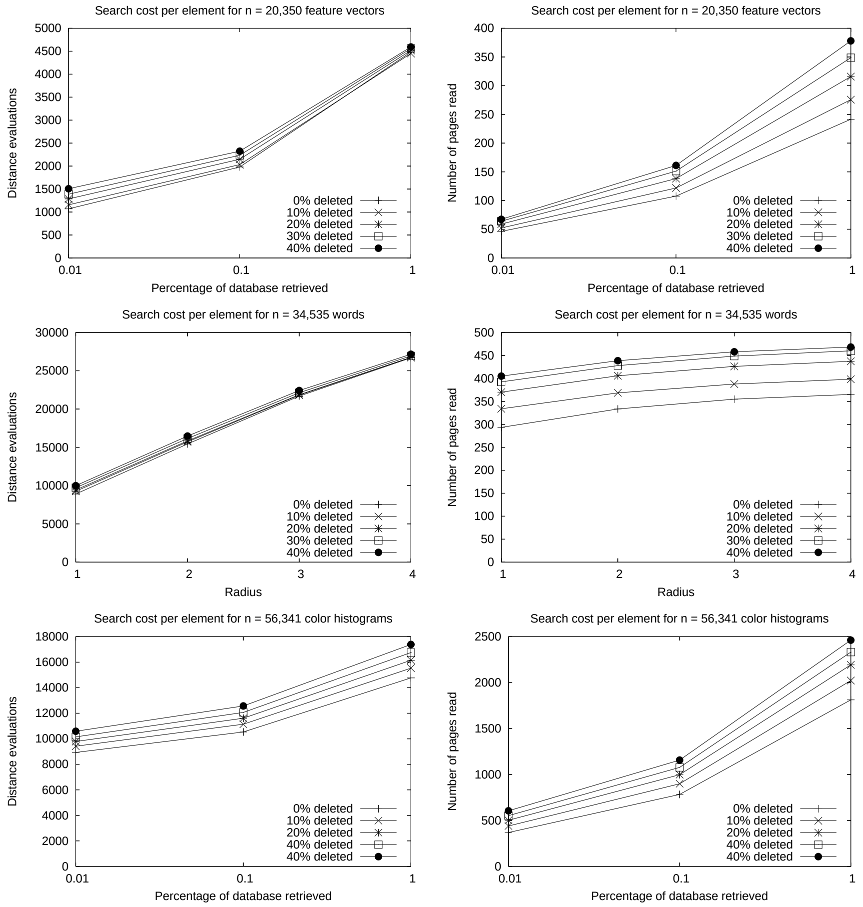


Figure 23: Comparison of search costs as the fraction of deleted elements in the database increases, for the best alternatives of the *DSC*.

the database. This degradation owes to two facts: deletions in the *DSAT* of centers can leave overestimated covering radii *CRZ*, and having more clusters with less elements reduces the value of each page read. Note that the former reason is not significant in the long term, since the in-memory *DSAT* is rebuilt each time the search engine is restarted.

7.3. Study of DSC on Larger Spaces

We have used relatively small datasets up to now. In this section we consider an order-of-magnitude larger datasets. First we study spaces of 1,000,000 uniformly distributed vectors, randomly generated in the real unit hypercube, with dimensions 10 and 15. We do not use the explicit coordinate information of the vectors, but just treat them as black-box objects with the Euclidean distance. These datasets are called *Vectors*. Second, we study a real space of 1,000,000

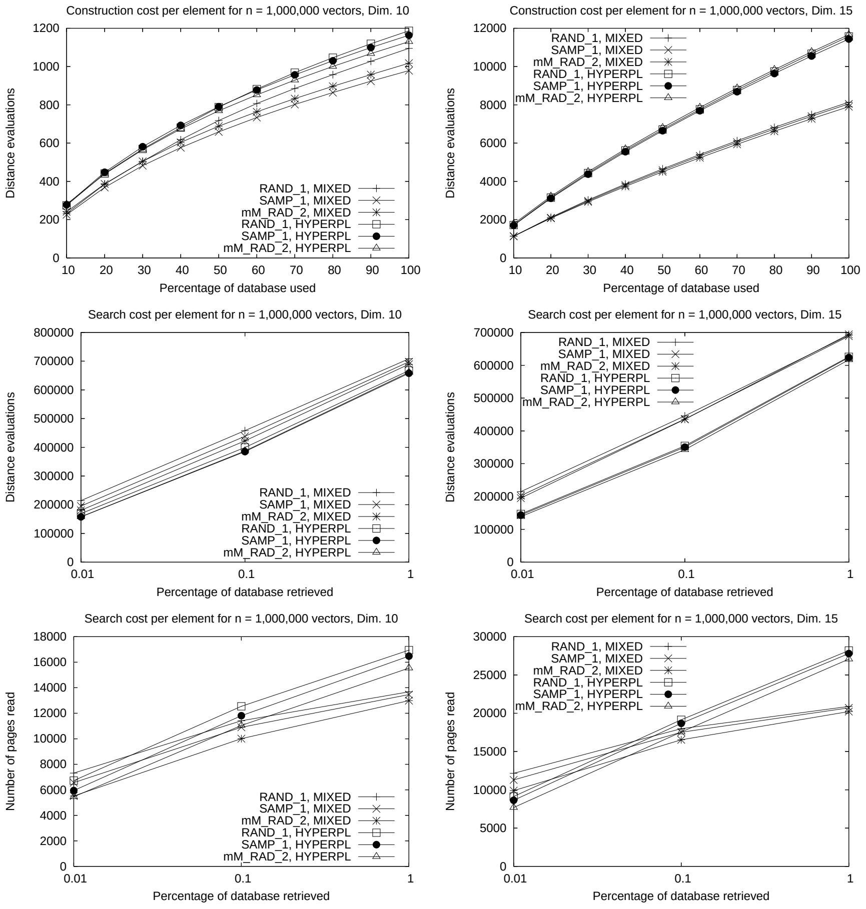


Figure 24: Insertion and search costs for the best alternatives of *DSC* on the space of vectors in dimensions 10 (left) and 15 (right), as a function of the policy used.

vector, which will be described later. On these larger spaces, we verify the conclusions obtained on *DSC*, the best structure that supports full dynamism (i.e., insertions and deletions). We also study a larger disk page size of 8 KB.

7.3.1. Uniform vector spaces

Figure 24 shows the construction and search costs in the *Vectors* space. Construction costs show the number of distance computations per element inserted; the number of I/Os is always 2–3 and is not shown. For the search costs we show both distance computations and number of pages read.

In these spaces, the effect of some groups of policies is more clearly visible. For example, the MIXED strategies yield better construction costs than the HYPERPL strategies, but also perform clearly worse at searches, in terms of distance computations. In terms of I/Os, the HYPERPL strategies are slightly

Dim	Alternative	Fill ratio		Total pages used	
		4KB	*8KB*	*4KB*	*8KB*
10	mM_RAD_2, MIXED	68%	69%	14,095	7,018
	mM_RAD_2, HYPERPL	54%	56%	17,755	8,553
15	mM_RAD_2, MIXED	67%	67%	21,272	10,548
	mM_RAD_2, HYPERPL	41%	42%	33,953	16,722

Table 2: Space usage for the space of vectors, with different page sizes.

better on more selective queries, but for less selective ones the MIXED strategies clearly take over. This behavior is more notorius as dimension grows.

Another noticeable effect is the sublinear growth of the insertion costs, for all the strategies. For example, in dimension 15, the MIXED strategies compare an object to be inserted with around 1.1% of the dataset when it contains 100,000 objects, but this percentage decreases to 0.8% when we have 1,000,000 objects. The HYPERPL strategies also show a decrease from around 1.9% to less than 1.2%. In dimension 10, the percentages with MIXED strategies go from 0.24% on 100,000 objects to less than 0.1% on 1,000,000 elements. With HYPERPL the percentages decrease too, from 0.28% to 0.11% approximately.

Figure 25 shows the effect of the disk page size, for the best strategies. We have used a disk page of 4KB throughout; now we consider the effect of doubling it to 8KB. The conclusions are not as simple as one might expect. For construction costs, since the in-memory *DSAT* has half the centers with pages of 8KB, the number of comparisons decreases (less than by half, as the search cost of the *DSAT* is sublinear). Besides, since in pages of 8KB there is space for more elements, the probability of a page split during an insertion decreases and so do the insertion costs.

At search time, instead, since the 8KB pages have more elements and those are all compared with the query, the number of distance computations increases with the disk page size (it does not double, since in exchange we read fewer disk pages, but not half). The number of disk pages read, instead, is almost exactly half when using the larger disk pages. Therefore, using larger disk pages does not necessarily improve search performance in distance computations, but it does in terms of I/Os. On the other hand, Table 2 shows that, both on dimensions 10 and 15, the fill ratios and space usage are unaffected by the change in disk page size. They also show that the MIXED strategy always makes a much better use of the space, as expected.

We also analyze deletion costs for the best alternative and different page sizes. Figure 26 shows deletions costs regarding distance computations and I/O operations. While a larger page size halves the number of distance computations, the number of I/Os stays at 2 for both dimensions. The reason for the reduced number of distance computations is that the in-memory *DSAT* has half the number of centers, and then deletions and reinsertions of centers, when these change, cost about half.

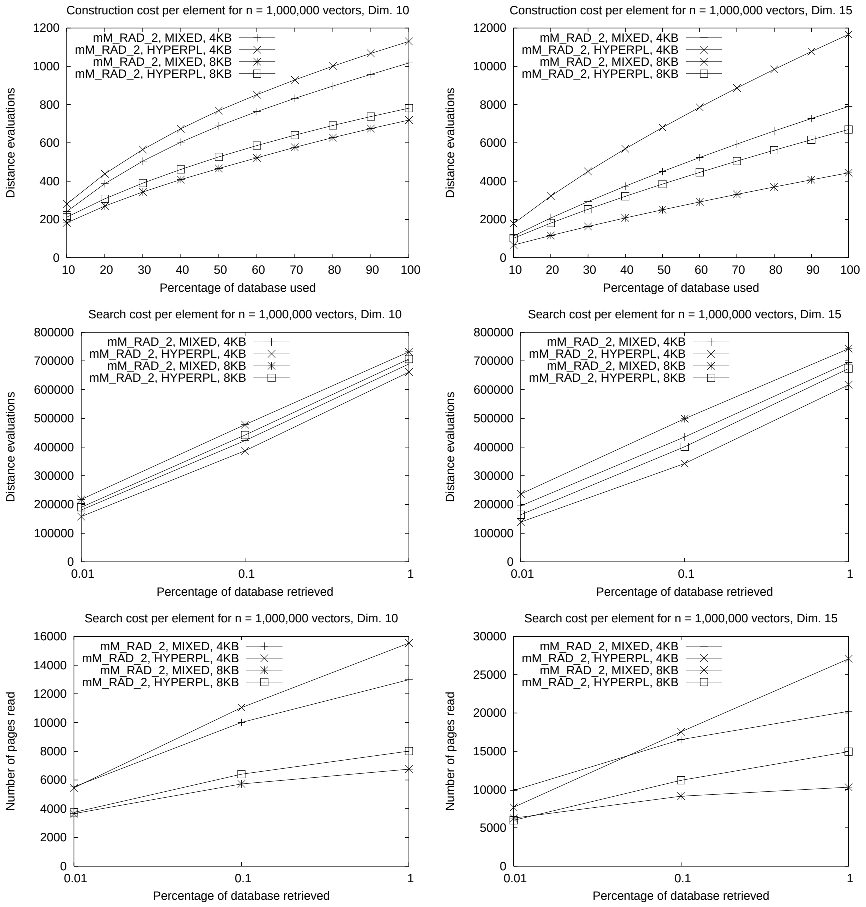


Figure 25: Insertion and search costs for the best alternatives of *DSC* on the space of vectors in dimension 10 (left) and 15 (right), as a function of the disk page size for the chosen policies.

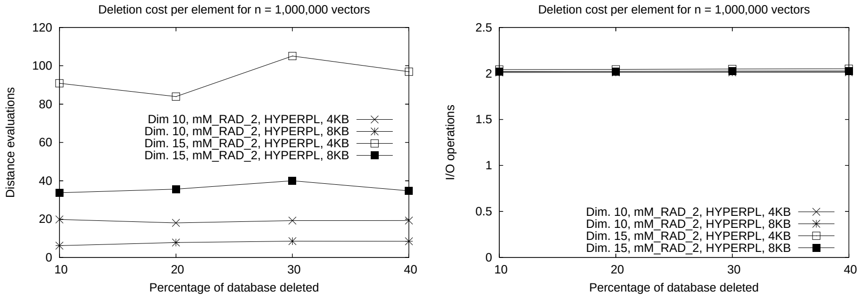
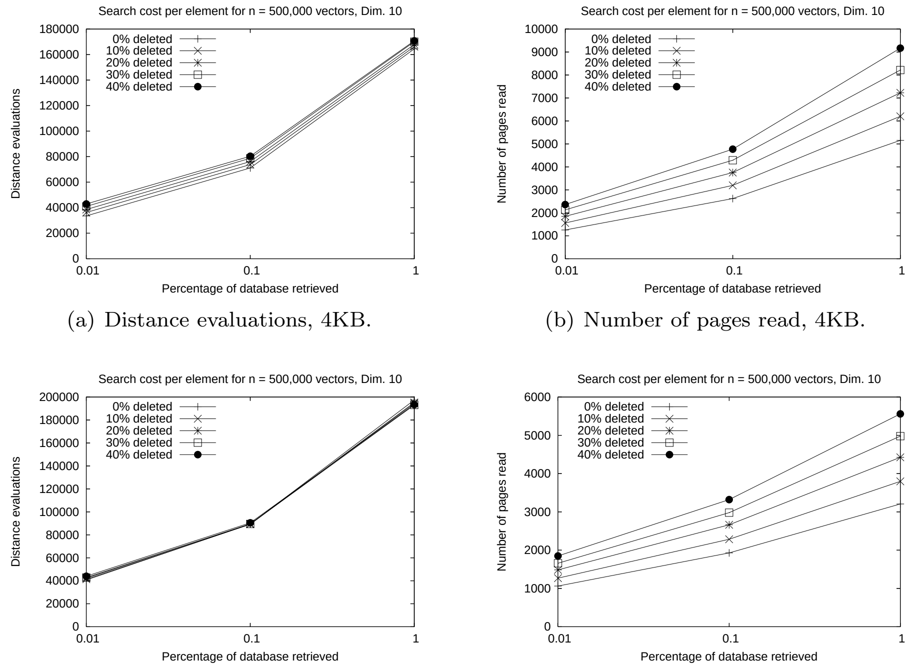


Figure 26: Deletion costs for the best *DSC* variant on the vectors in dimension 10 and 15.



(a) Distance evaluations, 4KB.

(b) Number of pages read, 4KB.

(c) Distance evaluations, 8KB.

(d) Number of pages read, 8KB.

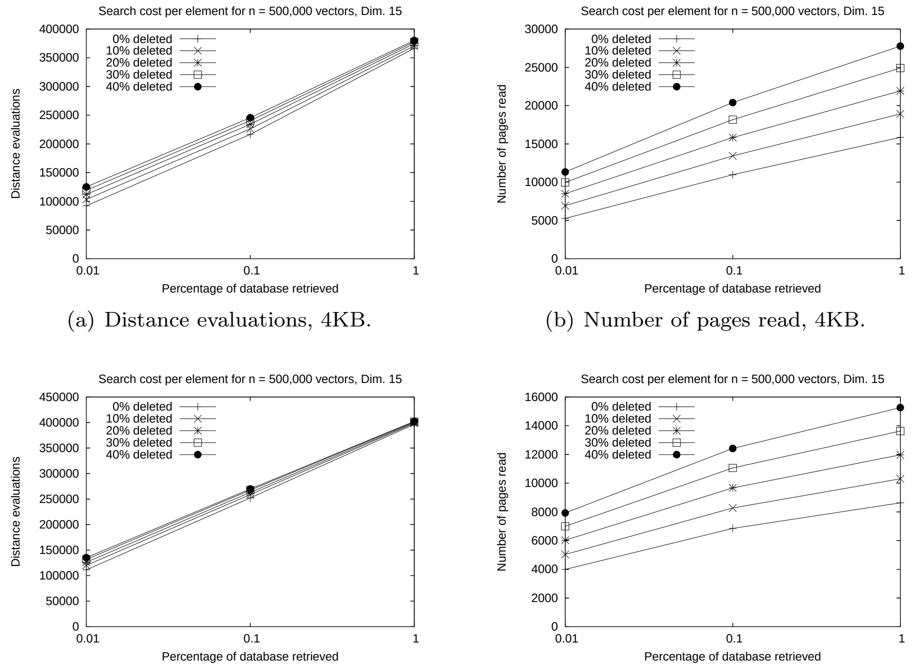
Figure 27: Comparison of search costs as the number of deletions grows, for the best alternatives of *DSC*, for vectors in dimension 10.

Finally, Figures 27 and 28 show how range searches are affected as the fraction of deletions increases, in dimensions 10 and 15, respectively, for both disk page sizes. It can be seen that, while the number of distance computations is basically unaltered, the number of I/Os increases steadily as the percentage of the database deleted grows up to 40%. The disk page size has almost no effect in the way the search times are degraded.

7.3.2. A real space of Flickr images

We test *DSC* on a real dataset of 1,000,000 image descriptors obtained from the SAPIR[8] image collection [6]. The content-based descriptors extracted from the images were: Color Histogram 3 × 3 × 3 using RGB color space (a 27-dimensional vector), Gabor Wavelet (a 48-dimensional vector), Efficient Color Descriptor 8 × 1 using both RGB and HSV color space (two 32-dimensional vectors), and Edge Local 4 × 4 (an 80-dimensinal vector). Therefore, each vector consists of 208 coordinates. The distance is Euclidean. This is a subset of CoPhIR [1], which contains around 106 million images from Flickr. We call

[8] Search In Audio Visual Content Using Peer-to-peer IR.



(a) Distance evaluations, 4KB.

(b) Number of pages read, 4KB.

(c) Distance evaluations, 8KB.

(d) Number of pages read, 8KB.

Figure 28: Comparison of search costs as the number of deletions grows, for the best alternatives of *DSC*, for vectors in dimension 15.

this database *Flickr*.

We compared the effect of the various policies for *DSC*, but we show only those that performed best. Figure 29 (left) shows the construction and search costs in this space. Construction costs show the number of distance computations per element inserted; the number of I/Os is always 2–3 and is not shown. For the search costs, we show both distance computations and number of pages read. As it can be observed, the MIXED partition strategies obtain better costs during insertions and, surprisingly, the center selection strategy of SAMP_1 with HYPERPL partition is the most expensive one.

The best alternatives are, in general, mM_RAD_2, RAND_1, and SAMP_1 for center selection, and pure (HYPERPL) or combined with balancing (MIXED) hyperplane distribution for partitioning. As expected, the balanced partitioning obtains again worse search costs than the others, because it prioritizes occupancy over compactness. Table 3 shows space occupancy.

For future comparisons we choose the variant mM_RAD_2 HYPERPL, as it has a good balance between distance computations and I/Os at searches. Its construction cost is not generally the lowest but it is reasonable.

We also measure deletion costs for the best alternative, with different page sizes. Figure 30 depicts deletions costs regarding distance computations and

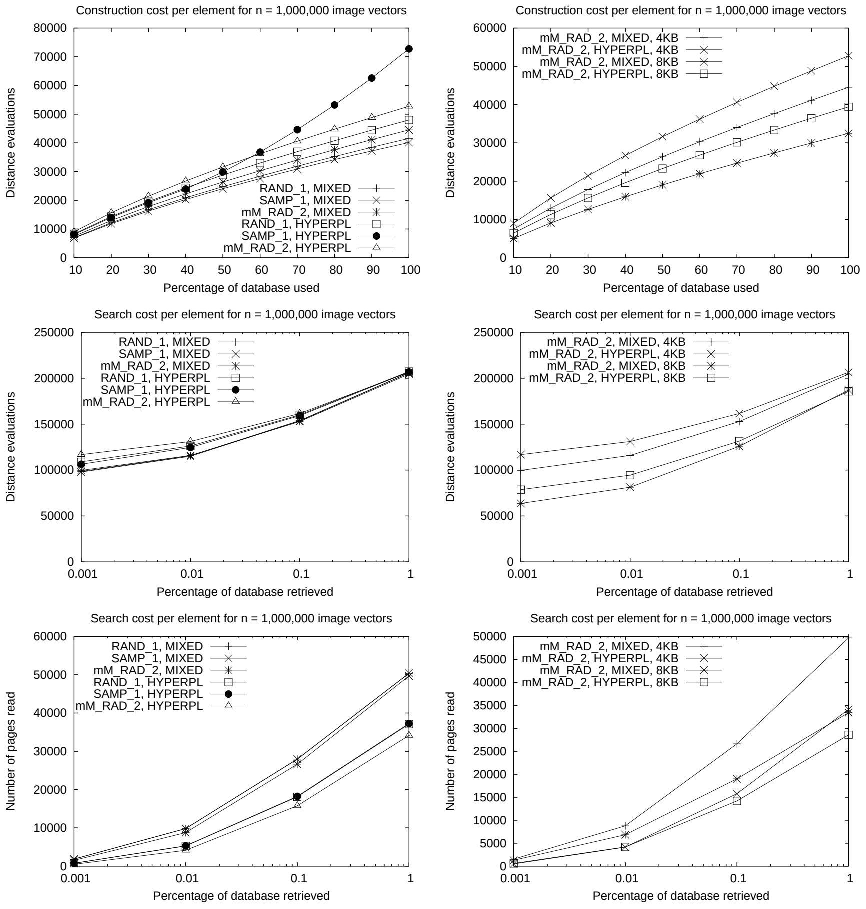


Figure 29: Insertion and search costs for the best alternatives of *DSC* on the space of *Flickr*. On the left, as a function of the policy used. On the right, as a function of the disk page size, for the chosen policies.

I/O operations. Again, while a larger page size halves the number of distance computations, the number of I/Os stays at 2. The reason for the reduced number of distance computations is the same as before.

Finally, Figure 31 shows how range searches are affected as the fraction of deletions increases, for both disk page sizes. It can be seen that, unlike what happens in the vector spaces, the number of I/Os is basically unaltered but the number of distance computations increases steadily with the percentage of deletions. Moreover, in this space the disk page size affects the way the search times are degraded: the number of I/Os grows more noticeably with page size of 8KB than with 4KB. The reason is that, with a larger page size, more pages with few elements are tolerated without removing them. For example, with a page size of 8KB the fill ratio decreases from 36.6% to 20.3% after the deletion

Alternative	Fill ratio		Total pages used	
	4KB	*8KB*	*4KB*	*8KB*
mM_RAD_2, MIXED	58%	56%	270,466	163,265
mM_RAD_2, HYPERPL	42%	37%	337,717	228,321

Table 3: Space usage for the *Flickr* dataset, with different page sizes.

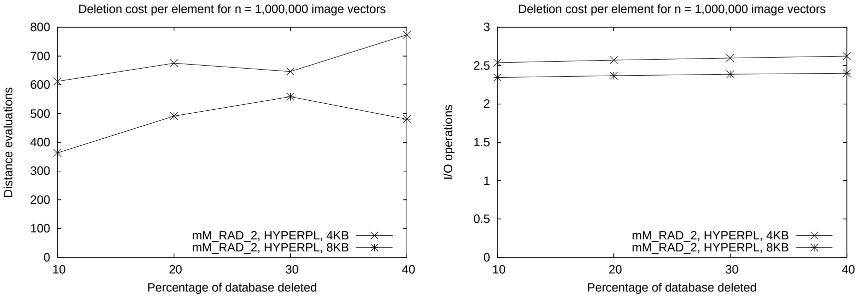


Figure 30: Deletion costs for the best *DSC* variant on *Flickr*.

of 40% of the elements, because 83% of the clusters are not removed.

7.4. Comparison with New Indices

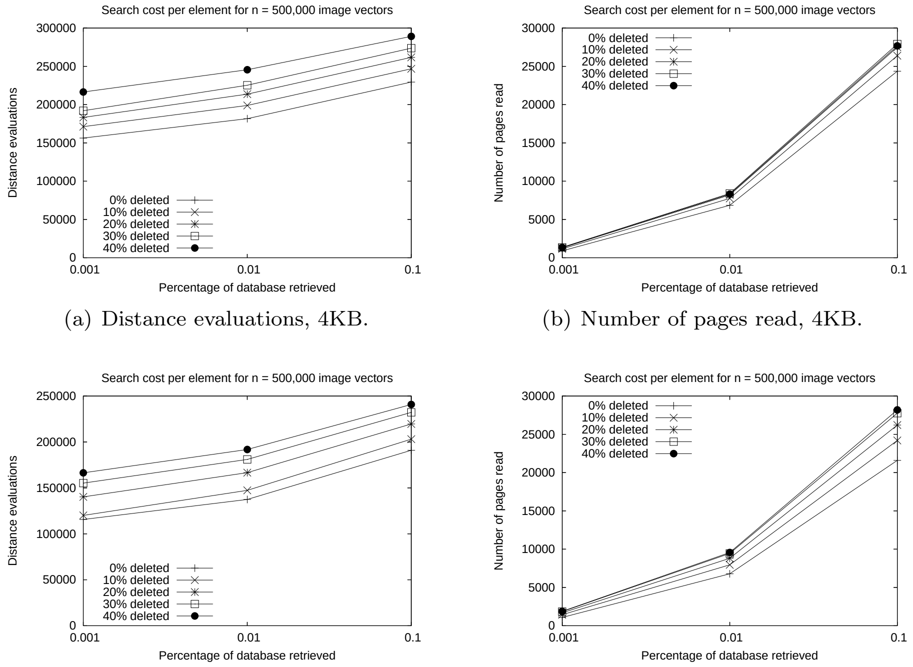
In this section we compare our index with other alternatives that are more recent than the M-tree, on the million-image Flickr space: the *eGNAT* [19] and the *MX-tree* [13].[9] We set their parameters as proposed by their authors for this space. For these experiments we use a page size of 4KB.

Figure 32 compares the construction and search costs of the *DSC* with the *eGNAT* and *MX-tree*, considering both distance computations and I/O operations. As it can be seen, the *DSC* is significantly more expensive at distance computations for insertions. However, it needs less than half of the I/O operations used by the *eGNAT* and approximately 10% of those used by the *MX-tree*. Once again, the extra work done by the *DSC* during insertions pays off at search time, since the *eGNAT* needs more than 5 times more I/Os and twice the distance computations of the *DSC*. The *MX-tree* is costlier at searches, regarding both distance computations and I/O operations. The reason is possibly that it uses copies of the elements as routing objects and this produces many additional I/Os. For the search experiments, the *MX-tree* index occupies 5,505,024 disk pages, whereas the *eGNAT* needs 393,216 and the *DSC* uses just 288,359 pages.

7.5. Scalability

Finally, we consider a significantly larger space to study how the performance of our best indices evolves with the size of the database. We generate synthetic

[9] The *MX-tree* code is available at `https://github.com/jsc0218/MxTree/`.



(a) Distance evaluations, 4KB.

(b) Number of pages read, 4KB.

(c) Distance evaluations, 8KB.

(d) Number of pages read, 8KB.

Figure 31: Comparison of search costs as the number of deletions grows, for the best alternatives of *DSC*, for image vectors of *Flickr*.

spaces in dimensions 10 and 15 as in Section 7.3, but now we generate 10,000,000 vectors. We build the index for increasing subsets of this dataset, and search for a fixed set of 1,000 random vectors with radii 0.7 and 0.65, respectively. This query retrieves at most 40 elements on the space in dimension 10 with the full dataset, and 725 on the space in dimension 15.

We compare the *DSAT+* with its best arities of 8 for dimension 10 and 16 for dimension 15, and the *DSC* with policy mM_RAD_2 HYPERPL, which performs better at searches despite being slightly more expensive to build. We use a page size of 8KB in this experiment.

Figure 33 shows the construction costs (note the logscale). While both insertion costs in terms of distance computations increase sublinearly, the growth rate of the *DSAT+* is much lower, raising just from 50 to 60 (in dimension 10) and 73 to 87 (in dimension 15) as we move from one million to ten million elements. The insertion costs of the *DSC* also grows sublinearly, despite being significantly higher: In 10 dimensions, it compares little more than 1,800 elements per insertion (0.18% of the dataset) when we insert one million elements, and ends up examining less than 6,000 (0.059%) when all the ten million elements are inserted. Similarly, in dimension 15, it starts examining 0.6% of the dataset and ends up examining 0.35%.

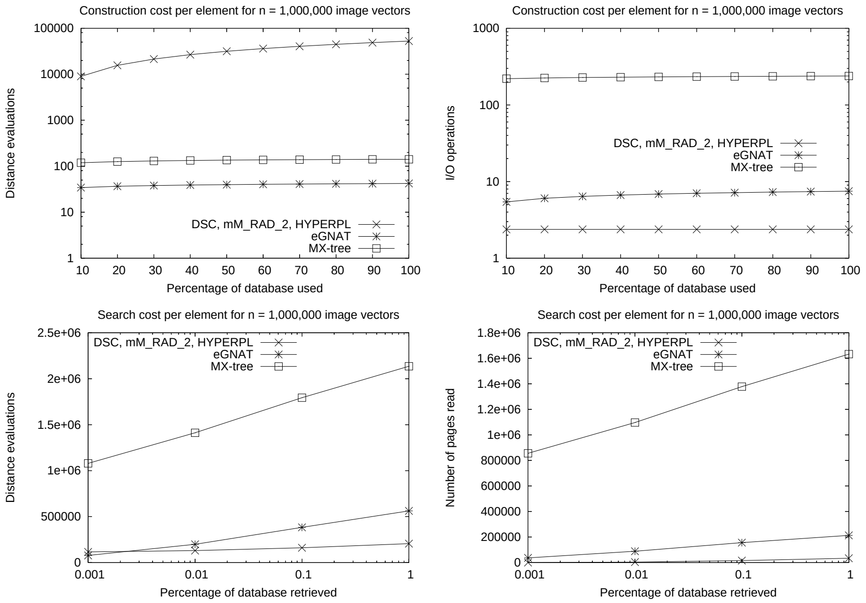


Figure 32: Comparison between *DSC*, *eGNAT*, and *MX-tree* on the space of *Flickr*.

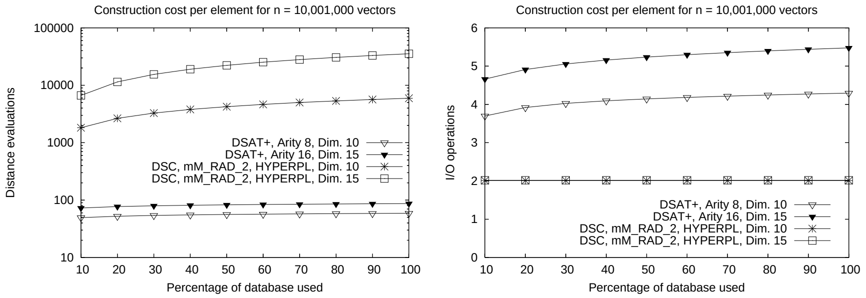


Figure 33: Comparison of insertion costs of the *DSAT+* and the best alternative of *DSC* on the space of 10,000,000 vectors.

In terms of I/Os, the situation is similar to what was observed in much smaller datasets: the *DSC* requires almost exactly 2 I/Os per insertion, whereas the cost on the *DSAT+* grows very slowly, from 3.7 to 4.3 and from 4.7 to 5.5, in dimensions 10 and 15 respectively.

Finally, Figure 34 shows how the search costs increase as the database size grows, in terms of distance evaluations and number of pages read. On searches, the *DSC* clearly outperforms the *DSAT+* in both aspects and in both spaces. For the space of dimension 15, for example, the *DSC* uses about half of the distance computations and a quarter of the I/Os needed by the *DSAT+*. Moreover, although the *DSAT+* performs better in spaces of lower dimension (or with

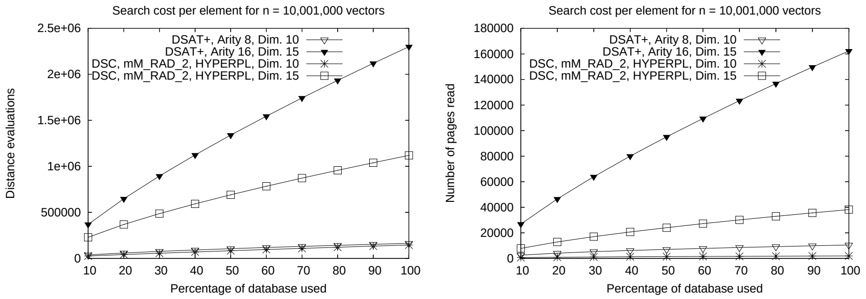


Figure 34: Comparison of search costs of the *DSAT+* and the best policy of the *DSC* on the space of 10,000,000 vectors, varying the database size.

more selective queries), the *DSC* performs slightly better in both aspects.

Note that, as the database size grows, the search times grow sublinearly in both indices and in both costs. For example, the *DSC* for the space of dimension 15 compares the query with 23% of the dataset with one million elements, but with just 11% with ten million objects. The fraction of disk pages read also decreases from about a half to less than 20%. In dimension 10, the *DSC* reads almost 8% of disk pages with one million elements, but only less than 2% with ten million elements. The percentage of objects compared with the query also decreases from almost 3% to less than 1.5%.

We remind that a space of dimension 15 is considered hard to index, and our results show that our indices will perform better as they handle larger spaces. This is in contrast with pivot-based indices, whose search cost grows linearly as the database increases (unless their space grows superlinearly with the database size, which becomes less and less acceptable with larger datasets).

8. Conclusions

We have presented three new dynamic metric indices for secondary memory. The *DSAT** and *DSAT+* structures extend an in-memory dynamic data structure [16] that offers a good balance between construction and search time. The secondary-memory version supports insertions and searches. The *DLC* extends an in-memory static data structure [2] that performs very well in spaces of medium and high dimension, but it is costly to build. The *DLC* supports insertions, deletions, and searches. It inherits both features of the static structure, thus its insertion time is very high in terms of distance computations, yet the number of I/Os is essentially 2. Finally, the *DSC* combines the *DLC* with the in-memory *DSAT* [16] in order to reduce the number of distance computations at insertion time.

The best variants of our structures generally outperform prominent alternatives in the literature like the *M-tree* [4], the *eGNAT* [19], and the *MX-tree* [13]. The *DSAT+* supports insertions with a few tens of distance computations and a few I/Os, and this cost grows very slowly with the database size. Its search cost is good when the spaces are of low to medium dimension or the queries are selective. The second structure, the *DSC*, outperforms the *DSAT+* at searches in terms of I/Os, and also in terms of distance computations when the spaces are of higher dimension or the queries have lower selectivity. At construction, the *DSC* requires 2 I/Os per insertion, but many more distance computations than the *DSAT+*, yet this cost also grows sublinearly.

Although our implementations are for proof of concept, and we could hardly index spaces significantly larger than those we have considered (ten million objects), our experiments show a marked sublinearity in the growth (or no growth) of both insertion, deletion, and search costs, both in distance computations and I/Os. This allows us to extrapolate that our indices would perform better on larger datasets. In contrast, indices based on pivots have linear-growing costs (because the probability of filtering an object at query time is fixed), unless their space per object is increased.

The most interesting future challenge would be to further reduce the number of distance computations required to insert elements in the *DSC*, which is its weakest point. Reducing the degradation suffered by searches when many elements have been deleted is also of interest, although this only occurs when a large fraction of the database has been deleted, and at this point a full reconstruction has a low amortized cost. Another challenge is to extend the *DSC* to cases where the part we maintain in memory (one object per disk page) does not fit in main memory either. In this case, a hierarchical structure (i.e., a *DSC* on the centers, instead of an in-memory structure) would allow handling extremely large datasets.

References

[1] P. Bolettieri, A. Esuli, F. Falchi, C. Lucchese, R. Perego, T. Piccioli, and F. Rabitti. CoPhIR: a test collection for content-based image retrieval. *CoRR*, abs/0905.4627v2, 2009.

[2] E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.

[3] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[4] 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.

[5] 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.

[6] F. Falchi, M. Kacimi, Y. Mass, F. Rabitti, and P. Zezula. SAPIR: Scalable and distributed image searching. In *SAMT (Posters and Demos)*, volume 300 of *CEUR Workshop Proceedings*, pages 11–12, 2007.

[7] K. Figueroa, G. Navarro, and E. Chávez. Similarity search and applications: Metric spaces library, 2007. Available at `http://www.sisap.org/Metric_Space_Library.html`.

[8] 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.

[9] 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.

[10] G. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580, 2003.

[11] G. R. Hjaltason and H. Samet. *Incremental Similarity Search in Multimedia Databases.* Computer Science Technical Report Series. Computer Vision Laboratory, Center for Automation Research, University of Maryland, 2000.

[12] 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.

[13] S. Jin, O. Kim, and W. Feng. MX-tree: A double hierarchical metric index with overlap reduction. In *Proc. ICCSA*, LNCS 7975, pages 574–589. Springer, 2013.

[14] M. Mamede. Recursive lists of clusters: A dynamic data structure for range queries in metric spaces. In *Proc. 20th ISCIS*, LNCS 3733, pages 843–853, 2005.

[15] G. Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.

[16] G. Navarro and N. Reyes. Dynamic spatial approximation trees. *ACM Journal of Experimental Algorithmics*, 12:article 1.5, 2009.

[17] G. Navarro and N. Reyes. Dynamic spatial approximation trees for massive data. In *Proc. 2nd SISAP*, pages 81–88, 2009.

[18] G. Navarro and N. Reyes. Dynamic list of clusters in secondary memory. In *Proc. 7th SISAP*, LNCS 8821, pages 94–105, 2014.

[19] G. Navarro and R. Uribe. Fully dynamic metric access methods based on hyperplane partitioning. *Information Systems*, 36(4):734–747, 2011.

[20] 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.

[21] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers Inc., 2005.

[22] T. Skopal, J. Pokorný, and V. Snásel. PM-tree: Pivoting metric tree for similarity search in multimedia databases. In *ADBIS (Local Proceedings)*, 2004.

[23] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, 2006.