Fully Dynamic Metric Access Methods based on Hyperplane Partitioning

Gonzalo Navarro[a], Roberto Uribe-Paredes[b,c]

[a] *Department of Computer Science, University of Chile, Santiago, Chile.* gnavarro@dcc.uchile.cl.
[b] *Departamento de Ingeniería en Computación, Universidad de Magallanes, Punta Arenas, Chile.* roberto.uribe@umag.cl.
[c] *Grupo de Bases de Datos - UART, Universidad Nacional de la Patagonia Austral, Río Turbio, Argentina.*

Abstract

Metric access methods based on hyperplane partitioning have the advantage, compared to the ball-partitioning-based ones, that regions do not overlap. The price is less flexibility for controlling the tree shape, especially in the dynamic scenario, that is, upon insertions and deletions of objects. In this paper we introduce a technique called *ghost hyperplanes*, which enables fully dynamic data structures based on hyperplane partitioning. We apply the technique to Brin's *GNAT* static index, obtaining a dynamic variant called *EGNAT*, which in addition we adapt to secondary memory. We show experimentally that the *EGNAT* is competitive with the *M-tree*, the baseline for this scenario. We also apply the ghost hyperplane technique to *Voronoi trees*, obtaining a competitive dynamic structure for main memory.

Keywords: Metric spaces, secondary memory, similarity search.

1. Introduction

Searching large collections for objects similar to a given one, under complex similiarity criteria, is an important problem with applications in pattern recognition, data mining, multimedia databases, and many other areas. Similarity is in many interesting cases modeled using metric spaces, where one searches for objects within a distance range or for nearest neighbors. In this work we call $d()$ the metric, which is nonnegative and satisfies the usual properties $d(x, y) > 0$ iff $x \neq y$, $d(x, y) = d(y, x)$, and the triangle inequality $d(x, y) + d(y, z) \geq d(x, z)$. We focus on *range queries*, where we wish to find every database object u such that $d(q, u) \leq r$, where q is our query object and r is the tolerance radius. Nearest-neighbor queries can be systematically built over range queries in an optimal way [1].

Metric access methods, *metric data structures*, or *metric space indexes* are different names for data structures built over the set of objects with the goal of minimizing the amount of distance evaluations carried out to solve queries. These methods are mainly based on dividing the space by using distances towards selected objects. Not making use of the particularities of each application makes them general, as they work with any kind of objects [2].

Metric space data structures can be roughly divided into those based on so-called *pivots* and those based on clustering the space [2]. Our work fits into the second category. These indexes divide the space into areas, where each area has a so-called *center*. Some data is stored in each area, which allows easy discarding of the whole area by just comparing the query with its center. Indexes based on clustering are better suited for high-dimensional metric spaces or larger query radii, that is, the cases where the problem is more difficult [2]. Some clustering-based indexes are the *Voronoi tree* [3], the *GNAT* [4], the *M-tree* [5], the *Slim-tree* [6], and the *SAT* [7].

These indexes are trees, where the children of each node delimit areas of the space. Range queries traverse the tree, entering into all the children whose areas cannot be proved to be disjoint with the query region.

There are two main tools to delimit areas in clustering-based metric structures: *hyperplanes*; and *balls* or *covering radii*. In the latter, clusters are delimited by radii around centers, thus the spatial shapes of clusters correspond to balls in an Euclidean space. In the former, closest to our work, the areas are defined according to the closest center to each point. The Euclidean analogous of this partitioning is the *Voronoi diagram* of the centers.

A problem with ball partitioning is that it is generally impossible to avoid that balls overlap in space. Therefore, even exact queries (with zero radius) usually have to enter several branches of the tree. In exchange, this overlap gives flexibility on where to insert new points, which has been key in techniques to maintain the tree balanced and to control the page occupancy in secondary memory. The *M-tree* is a paradigmatic example of these techniques. The *Slim-tree* was designed as a way to reduce the overlaps of the *M-tree* through periodic reorganizations. In contrast, no overlaps are possible in hyperplane-based partitioning methods. While this gives an advantage for queries, these methods are more rigid and controlling the tree shape is difficult. The *GNAT* and the *SAT* are good exponents of this trend.

Most structures for metric spaces are not designed to support dynamism, that is, updates to the database once the index is built [2]. Yet, some structures allow non-massive insertion operations without degrading their performance too much. In recent years, indexes with full dynamic capabilities (that is, allowing massive updates) have appeared, such as the *M-tree* [5, 8] and a dynamic *SAT* [9]. Due to the rigid nature of the structures, handling updates have proved to be more complex on hyperplane-partitioning indexes, especially to guarantee that the data structure quality is not degraded after massive updates. The most serious problem is how to replace a deleted center with another while ensuring that all the existing subtrees are covered by the correct areas (this can be achieved with ball partitioning structures by simply enlarging the covering radii, but no such flexibility exists with hyperplanes). In the dynamic *SAT* deletions are handled via local reconstructions, which is expensive [9].

Implementation in secondary memory is also complex. To begin with, the number of distance computations competes in relevance, as a cost measure, with the I/O access cost. Most of the structures for metric space search are designed for main memory. At present there are a few dynamic metric indexes for secondary memory. The best known is the *M-tree*. Others, based on pivots, are the OMNI methods [10] and the iDistance [11]. To our knowledge, no hyperplane-partitioning based metric index exists for secondary memory that supports insertions and deletions (a recent variant of the *SAT* [12] handles insertions and searches). The difficulty, added to that of handling deletions in general, is that there is no control over the insertion positions of new elements. Thus it is hard to maintain those trees balanced, and to ensure an adequate space utilization at the leaves. A poor space utilization not only yields larger space usage, but also more disk accesses to reach the data.

In this paper we face those challenges, by introducing a novel technique called *ghost hyperplanes* that deals with the problems of rigid structures. We apply the technique to the *GNAT* data structure [4], obtaining a dynamic variant we call *EGNAT*. The quality of the *EGNAT* structure is maintained upon insertions and deletions, while keeping their cost low. The result is shown experimentally to be superior to the *M-tree* in various scenarios, thanks to the intrinsic advantage given by the nonoverlapping areas. To illustrate the generality of the technique, we apply it to another rigid structure, the *Voronoi tree* [3]. The result is competitive with the *EGNAT* in main memory.

The paper is organized as follows. In Section 2 we describe the *EGNAT*, focusing only on the data structure, insertion and search operations. Section 3 introduces the ghost hyperplane technique and applies it to delete elements from the *EGNAT*. It also evaluates the data structure experimentally in main memory. Section 4 extends the *EGNAT* to secondary memory, and compares it experimentally with the *M-tree*. Section 6 applies the ghost hyperplanes idea to Voronoi Trees, and gives experimental results as well. Finally, we conclude and give future work directions in Section 7.

2. *EGNAT*

The *EGNAT* is based on the concept of hyperplane partitions. It is a variant of the *GNAT* index proposed by Brin [4]. The *GNAT* is a k-ary tree built by selecting k centers (database objects) at random, $\{p_1, p_2, \ldots, p_k\}$, which induce a Voronoi partition of the space. Every remaining point is assigned to the area of the center closest to it. The k centers form the children of the root and the points assigned to their areas recursively form subtrees, D_{p_i}. Each child p_i maintains a table of distance ranges toward the rest of the sets D_{p_j}, $range_{i,j} = (\min\{d(p_i, x), x \in D_{p_j}\}, \max\{d(p_i, x), x \in D_{p_j}\})$. See Figure 1.

The *EGNAT* contains two types of nodes, *buckets* (leaves) and *gnats* (internal nodes). Internal EGNAT nodes are similar to internal GNAT nodes, whereas leaves contain a number of elements for

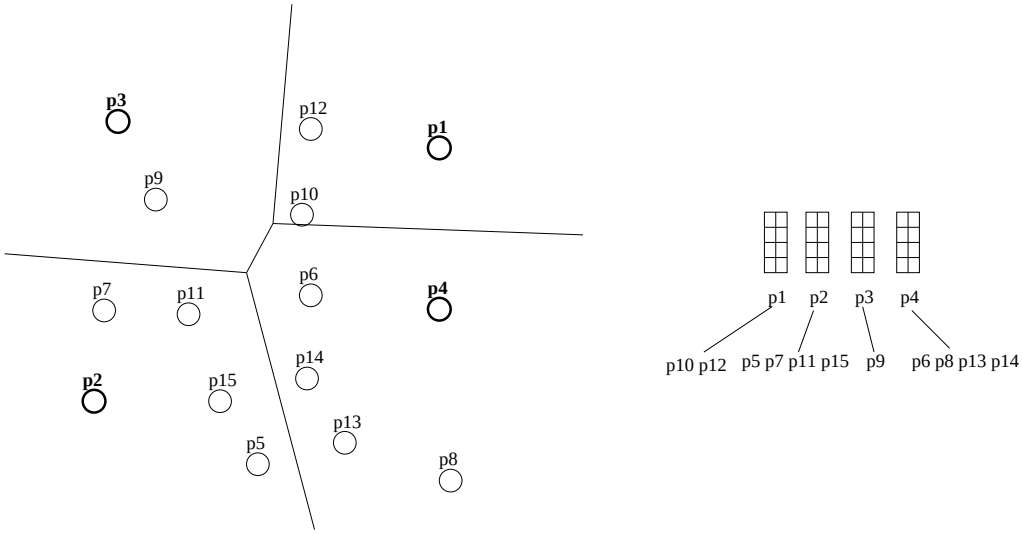

Figure 1: *GNAT*: partition of the space and representation of children and range tables.

which only the distance to their parent is known. Nodes are initially created as *buckets*, without internal structure. This yields a significant reduction in space compared to the GNAT, while still allowing some pruning at query time by applying the triangle inequality using the distance to the parent when scanning the *bucket*. The bucket size yields a space/time tradeoff.

Insertions proceed by recursively choosing the closest center for the new element, until reaching a leaf, where the element is inserted in the *bucket*. When a *bucket* becomes full, it evolves into a *gnat* node, by choosing k centers at random from the bucket and re-inserting all the other objects into the newly created node. New children of this *gnat* node will be of type *bucket*.

Thus, the structure is not built in a bottom-up manner, as an *M-tree*, for example. It is built in top-down form, except that leaves are converted into internal nodes only when a sufficiently large number of elements have been inserted into them. Yet, their splits do not propagate higher in the tree.

The range search algorithm for query q and radius r proceeds as follows. For *gnat* nodes, the query uses the standard recursive *GNAT* method, shown in Algorithm 1, until reaching the leaves[1]. When we arrive at a bucket node, which is the child corresponding to a center p of some internal node, we scan the bucket, yet use the triangle inequality to try to avoid comparing every object x in the bucket: If $|d(x,p) - d(q,p)| > r$, then we know that $d(x,q) > r$ without computing that distance. We note that $d(q,p)$ is already computed when we arrive at the node, and that distances $d(x,p)$ are those stored in the bucket together with the objects.

Algorithm 1 range search at a *gnat* node.

rangesearch(Node P, Query q, Range r)

```
1: c ← argmin_{1≤i≤k} d(p_i, q)
2: d_min ← d(p_c, q)
3: for all p_i ∈ P do
4:    if [d_min − r, d_min + r] ∩ range_{i,c} ≠ ∅ then
5:       if d(p_i, q) ≤ r then
6:          Report p_i
7:       end if
8:       rangesearch(D_{p_i}, q, r)
9:    end if
10: end for
```

3. Deleting in the *EGNAT*: Ghost Hyperplanes

Deleting elements is the most serious problem we face in the design of a fully dynamic data structure. Deleting from a bucket is of course simple, but deleting an internal node is complicated because there is no chance of finding a replacement with its same area boundaries.

[1] The real *GNAT* algorithm is more complex, as it avoids comparing the query with some centers, by using an AESA-like [13] algorithm over the zones. The *EGNAT* uses the simpler method we present here.

One path explored in the past [9] is fully reconstructing the subtree rooted at the deleted element, but this has shown to be too expensive, so we aim at a more lightweight technique.

Another "solution" indeed used by some classical database managers is to simply mark the deleted object and retain it for indexing purposes. While this is certainly an option when indexing scalar data such as numbers and strings, it is not that acceptable in the metric space scenario. First, the indexed objects can be very large (images, audio, or video streams, complex meshes, etc.). In cases where massive deletions occur, the idea of a very large database using a significant percentage of extra space due to the lack of a deletion algorithm is hard to defend. One can reconstruct it periodically, thus falling down again to carrying out expensive reorganizations every short periods (longer periods imply a higher percentage of wasted space).

Second, because the objects are not "anonymous" scalars but complex entities, there may be even legal barriers to maintaining them in the database. Imagine for example a database of online music records that dynamically, upon changes in the contracts with suppliers, must add and remove records. It might be simply not possible, for copyright reasons, to maintain deleted records for indexing purposes. The same can happen with medical records (X-ray images, for example) of patients, or biometric information of the employees of a company, or fingerprints at police databases, etc. for privacy reasons, depending on the law.

Finally, one could aim at using synthetic objects for indexing purposes, but these are again a waste of space, and the ability of generating a valid object that is in addition a good representative of a set is highly dependent on the specific metric space, and thus not applicable as a general technique.

Thus facing the algorithmic challenge of carrying out actual deletions in a hyperplane-based metric index is relevant, especially if the alternative of ignoring the problem can be far from satisfactory from various points of view. In this section we describe and evaluate an innovative solution to this problem, called *ghost hyperplanes*. As we will show later, this method sometimes performs better than just marking deleted elements without removing them, which gives yet another justification to study the problem.

We note, before entering into details, that nodes never become of type *bucket* again once they have been promoted to *gnat*, even if deletions would make a *gnat* subtree fit in a bucket. The reason is that there is a considerable cost for converting a bucket of size m into a *gnat* node ($O(km)$ distance computations, to find the subtree of each object and to compute the $range_{i,j}$ tables). Periodic subtree reconstructions (much sparser than those necessary when no deletions are carried out, as we will see in the experiments) handle this problem.

3.1. Ghost Hyperplanes

The method consists in replacing the deleted object x by another, y, which will occupy the placeholder in the node of x *without modifying the original ranges of the deleted object*. The node that held x is then marked as *affected*, and the distance between x and y is recorded in the node.

As we replace the center x by y, the partition implicitly changes its shape, altering the hyperplanes that separate the current region with the neighboring ones (note that the hyperplanes themselves are never stored, but instead implicitly defined by the centers). Furthermore, there is no re-calculation of $range_{i,j}$. This has to be accounted for at the time of searching, insertion, and deletion in the *EGNAT*. This method is called *ghost hyperplanes* because the old hyperplanes, which have disappeared after the replacement, will still influence the behavior of the data structure.

Figure 2 illustrates this concept. The old hyperplanes, called the "ghost" ones, correspond to the deleted object. The nodes already inserted in the affected subtree have followed the rule of those ghost hyperplanes. The new hyperplanes are the real ones after the deletion, and will drive the insertion of new objects. That is, insertions will consider the new objects, and therefore the new elements inserted into the affected subtree will be those closest to the new element.

However, we must be able of finding the elements inserted before *and* after the deletion. For this sake, we must introduce a *tolerance* when entering into affected subtrees. Say that node p has been deleted and replaced by node p'. Let $I = d(p, p')$ be the distance between the original element and its replacement. Then, since we will only store the new element as the center, but still want to find those elements inserted before doing the replacement, we must consider that the hyperplanes may be off up to I. Therefore, to implement ghost hyperplanes, the only extra information we store on *gnat* nodes v is the tolerance I_v. If $I_v = 0$, the node has not been affected by replacements.

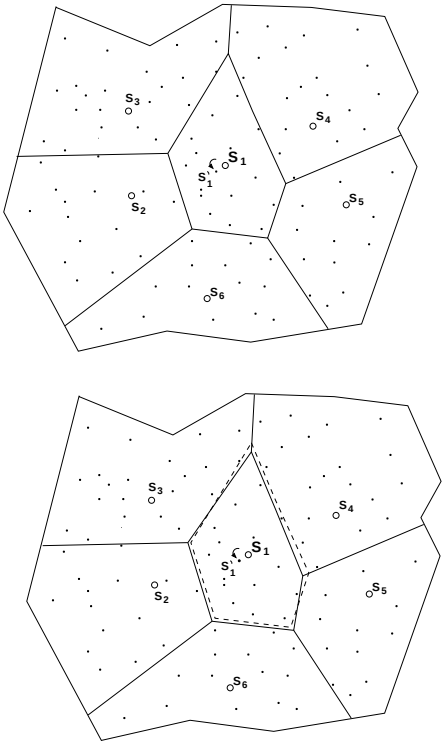

Figure 2: Partition of the space before and after the deletion of center s_1, which is replaced by s'_1. The new hyperplanes are drawn with solid lines, but the old ("ghost") ones, in dashed lines, were used when inserting old elements, and this must be considered in order to find those elements later.

In particular, for range searching we must modify line 4 of Algorithm 1, so that if node p_i is affected with distance I_i, we use $[d_{min} - I_i - r, d_{min} + I_i + r]$, instead of just $[d_{min} - r, d_{min} + r]$. Moreover, if the nearest center c is also affected with distance I_c, we must use tolerance $I_i + I_c$. Only if the more tolerant intersection is empty we can safely discard the subtree in the search process. Also, the I value of the parent must be considered when filtering the nodes in the child *bucket* using the triangle inequality (as they store the distance to their parent center).

In addition, we must take care about further deletions. Imagine that, after replacing p by p', we now delete p' and replace it by p''. Now the hyperplanes may be off by up to $d(p, p') + d(p', p'')$. In general, all the *gnat* nodes v will maintain an I_v value, initially zero, and upon each replacement on the center of v, I_v will be *increased* by the distance between its deleted center and its replacement. We might rebuild a subtree when its I_v values have reached a point that queries become too inefficient.

3.2. Choosing the Replacement

Election of the replacement object is not obvious. On one hand, we wish that the replacement is as close as possible to the deleted object, to minimize the increase of I. On the other, we wish to find it with as low cost as possible.

We note that, prior to choosing the replacement, we must find the node to be deleted in the tree. This is done via an exact search (i.e., with radius zero). Then we must choose a replacement object, for which we devise at least the following alternatives, from most to least expensive.

1. The nearest neighbor. A natural solution is to choose the nearest element in the dataset as a replacement. This can be found by a nearest neighbor query, whose nonnegligible cost must thus be added to that of the deletion. As this replacement needs not be in a leaf, migrating it to the deleted node triggers a new deletion subproblem. Moreover one must avoid cycles in the sequence of nearest neighbors found, which further complicates the problem.

2. The nearest descendant. A way to guarantee termination is to choose the nearest neighbor from the descendants of the node where the deleted object lies. This limited nearest neighbor search is likely to produce a good candidate, as we search within the partition of the query element. Still, a single deletion will usually introduce several ghost hyperplanes, which is likely to be worse than making just one replacement with a not-so-close element in the first place.

3. The nearest descendent located in a leaf. A third alternative is the replacement by the nearest descendant of the deleted object, among those located in a *bucket*. This might produce larger I values than previous alternatives, but it guarantees that only one nearest object search will be carried out, and that no cascading eliminations (nor creations of further ghost hyperplanes) will arise.

4. A promising descendant leaf. To avoid running the expensive nearest-neighbor procedure, we might just descend to the leaf where the element to delete would be inserted, that is, choose the closest center at each point. Then the closest object at the *bucket* is chosen.

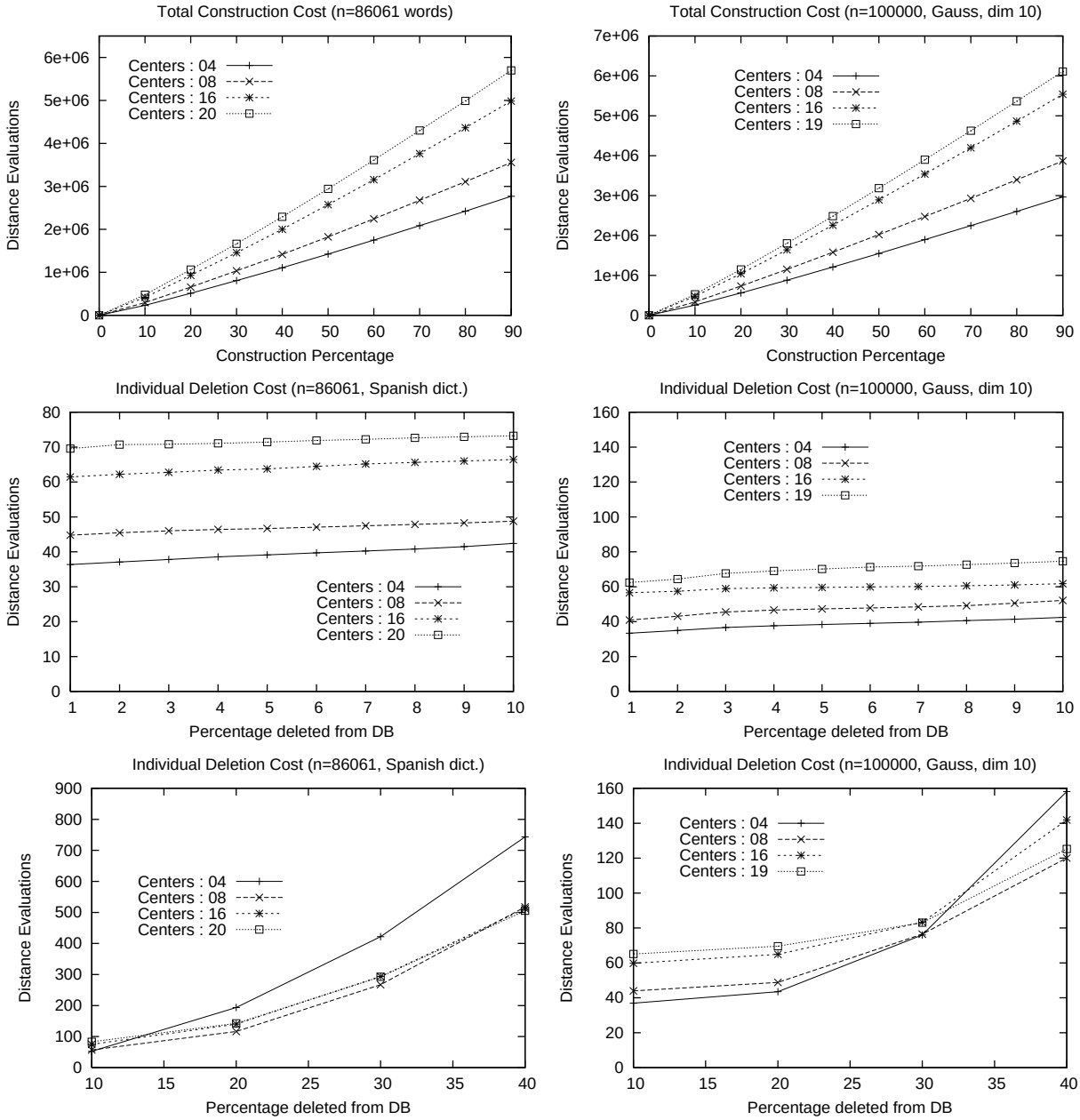

Figure 3: Aggregated construction costs (top) and individual deletion costs when deleting the first 10% (middle) and 40% (bottom) of the database. The different lines refer to different tree arities (centers per node).

5. An arbitrary descendant leaf. The least sophisticated alternative is the direct replacement of the deleted element by an arbitrary object at a descendant leaf. This can cause an even larger overlap, but there is no cost in terms of distance evaluations.

3.3. Experimental Evaluation

Tests made in two spaces from the Metric Spaces Library (`www.sisap.org`) were selected for this paper. The first is a Spanish dictionary with 86,061 words, where the edit distance is used. This is the minimum number of insertions, deletions or substitutions of characters needed to make one of the words equal to the other. The second is a synthetic 10-coordinate Euclidean space with Gaussian distribution with mean 1.0 and variance 0.1, containing 100,000 points. We consider *EGNATs* of arities $k = 4$ to $k = 20$. Bucket sizes are 102 for the strings and 92 for the vectors, to match later disk page sizes. We create the *EGNATs* with 90% of the dataset, and reserve the rest for queries. The effect of deletions is shown by deleting and reinserting 10% to 40% of the *EGNAT* after it is built. The strings are searched for ranges 1 to 4, whereas the vectors are searched with radii that recover, on average, 0.01% to 1% of the database.

We first show aggregated construction costs in Figure 3 (top), via successive insertions. For the Spanish dictionary, the insertion cost per element ranges from around 40 to 80 distance computations, depending on the arity of the tree. For the Gaussian space, the range goes from 33 to 66.

The height of the trees is always between 4 and 5, despite the lack of balancing guarantees. Thus the variance in the searches is always small. On the other hand, around 6% of the nodes are of type *gnat* in both cases; the rest are *buckets*.

Individual deletion costs are also shown in Figure 3. We are using the fifth replacement policy, for simplicity. When we have deleted a small part of the database (10%, middle), the deletion costs are close to insertion costs. As we pay no distance evaluations to find the replacement, the deletion cost is simply the cost of a search with radius zero. This should indeed be similar to an insertion cost (which goes by a simple branch towards a leaf) when there are no ghost hyperplanes. However, after a massive deletion (40%, bottom), the ghost hyperplanes make the zero-radius search much more expensive, and consequently we can see that deletion costs deteriorate considerably.

We now study replacement policies for deletion. We show that the fifth policy, despite of its simplicity, is competitive with more complex ones. For this sake we compare it with the third, which is likely to produce a very good candidate and is still reasonably simple.

Figure 4 shows the relative difference between both methods (a positive percentage indicates that the third alternative is better), considering the cost of deletions and that of searches after the deletions, respectively. As it can be seen, the difference is always very small. This is explained by the fact that most of the elements are very close to (or at) the leaves, and therefore either the policies do not play any role, or all the candidates are well clustered so that finding the closest one is not too different from finding a random one.

For searching, the third policy is usually better, as expected, since the ghost hyperplanes are tighter. For deleting, at first the fifth method is better because it is cheaper, yet in the long term (larger percentages of the DB deleted) the higher cost of finding the element to delete dominates. Since the differences are anyway very small, we have chosen the fifth method for simplicity.

Figure 5 shows the search cost just after construction, and after we delete and reinsert 10% and 40% of the database. Interestingly, the effect of deletions is not so sharp on proximity searches (with radius larger than zero), as these searches have to enter many branches anyway.

The results show that the methods are very robust in maintaining the quality of the database. Full reconstruction of subtrees is necessary only after a massive amount of deletions and reinsertions have been carried out on them.

4. Secondary Memory

In secondary memory we account not only for the distance evaluations, but also for the disk reads, writes, and seeks. A secondary memory version of *EGNAT* fits each node and each leaf in a disk page.

In the experimental results, we assume the disk page is of size 4KB, which determines arity $k = 20$ (dictionary) and $k = 19$ (vectors) for the *EGNATs*. To understand this low arity, recall that we need to store a quadratic number of $range_{i,j}$ limits. Many more elements fit in the leaves.

We experimentally evaluate the *EGNAT* and compare it with the baseline for metric space

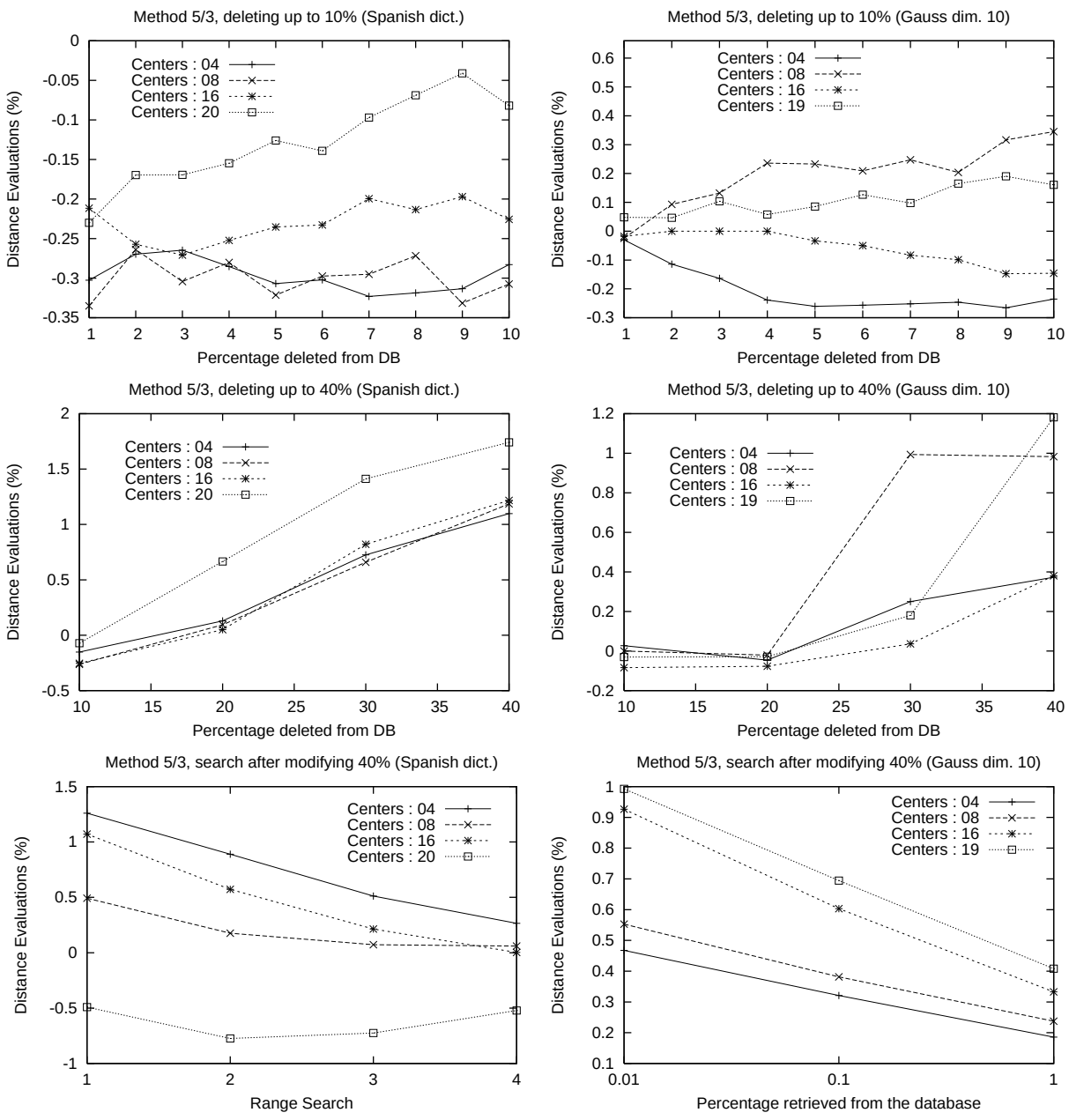

Figure 4: Ratio between methods, in deletion (top, middle) and search (bottom) costs.

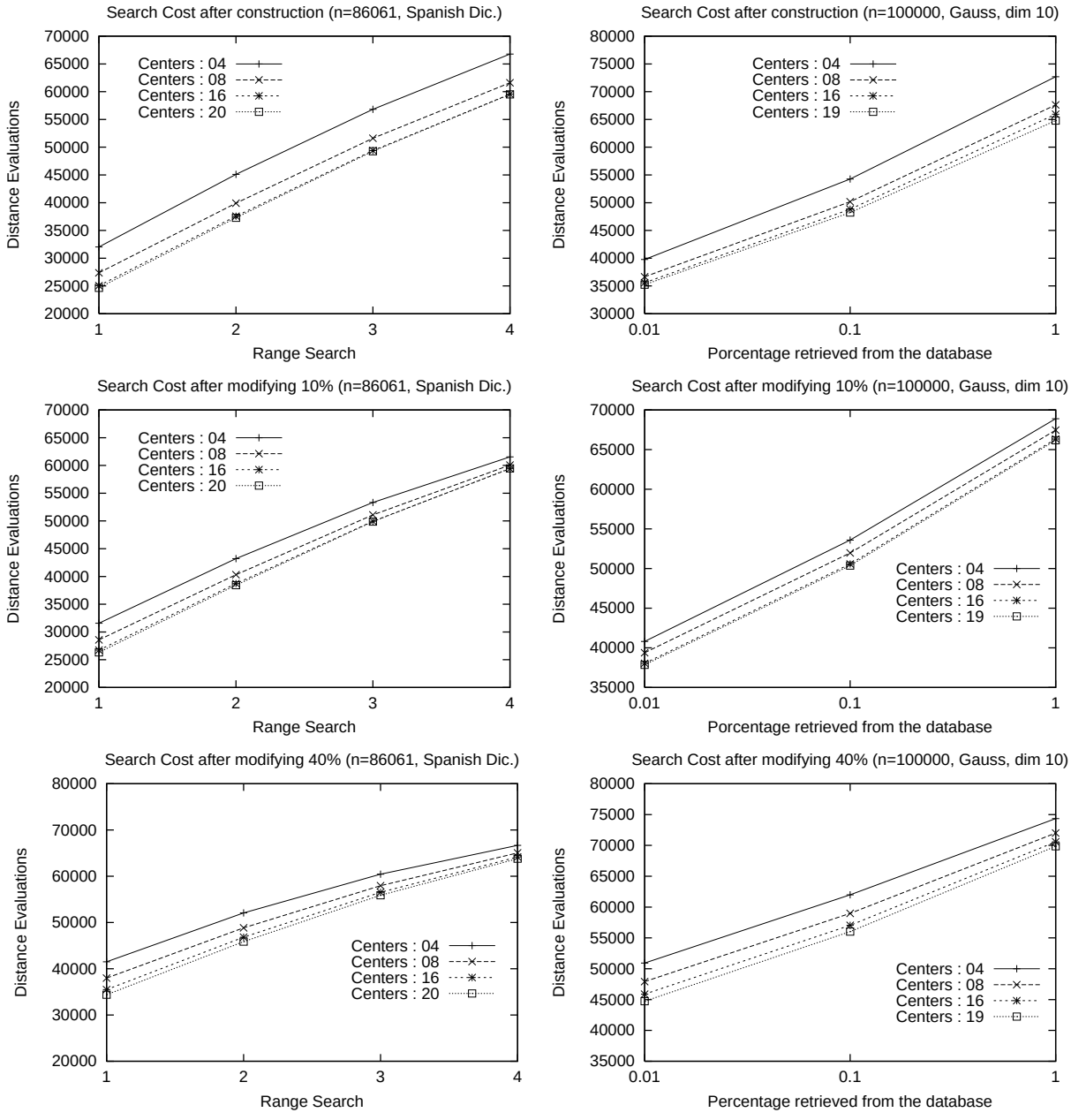

Figure 5: Search costs after construction and after deleting and reinserting part of the database.

search in secondary memory: the *M-tree* [5]. Deletions are not implemented in the standard *M-tree* distribution we use for comparison (at `http://www-db.deis.unibo.it/Mtree`), so in this case we only test the *EGNAT*. The parameters chosen for the *M-tree* are G_HIPERPL (which gave us the best results) and MIN_UTIL = 0.1 and 0.4, to test two page fill ratio guarantees [14]. For other potential competitors [10, 11] we could not find the code publicly available, and furthermore they do not report results on deletions either.

A serious problem with the *EGNAT* is its poor disk space utilization. While the *M-tree* achieves an average page fill ratio of 48%-56% on strings and 25%-30% on vectors (depending on MIN_UTIL), the *EGNAT* achieves 19% and 18%, respectively. The problem is that, with large arities (19 and 20), bucket pages are created with few elements (5% utilization), and this lowers the average page usage. A natural solution is to share disk pages among buckets, that is, the bucket size is set to $1/k$ of the physical disk page, which is shared among k buckets. By sharing, for example, disk pages among $k = 6$ buckets, the disk page utilization raises to 25% on both spaces. For $k = 4$, utilization is 22%. We will call this latter alternative *EGNAT-B*.

Aggregated construction costs are shown in Figure 6. Both structures have similar construction cost in terms of distance evaluations and disk writes (*EGNAT-B* being worse in the latter), yet the *M-tree* does many more disk block reads. This owes to the effort made in the *M-tree* insertion algorithms to achieve a balanced partition of the objects across disk pages. Therefore, the price of the better space utilization of the *M-tree* is a higher number of disk reads at insertion time. The number of disk writes, instead, grows slower on the *M-tree*, partly due to the better space utilization.

Figures 7 and 8 (top) compare search costs, in terms of distance evaluations and number of reads or seeks (the number is the same for both). The *EGNAT* performs fewer distance computations (as little as one half for selective queries), yet the *M-tree* carries out (sometimes significantly) fewer I/O accesses. The *EGNAT-B* pays a significant extra cost in terms of I/Os. The outcome of the comparison, in general, will depend on the cost of distance computations in a particular metric space, compared to the disk access cost.

Figures 7 and 8 (middle) show deletion costs in terms of disk block reads and writes, for the *EGNAT*. It is interesting that writes are always upper bounded by 2, whereas reads do degrade as more elements are reinserted. This is because the number of pages written is always bounded in the algorithm, yet disk reads are needed to find the element to delete. This read cost would be significantly lowered if one had a handle indicating where is the element to delete located in the tree.

Finally, the bottom rows of the figures show the search costs, in terms of distance computations and disk accesses, after a portion of the database is deleted and reinserted. We experiment here with a variant called "marked", where deleted elements are just marked as such without actually removing them from the tree. The search is as usual except that marked elements are never reported. As explained, this is not an acceptable deletion method in metric spaces, but it serves as a baseline to see how much the search deteriorates due to the ghost hyperplanes. It is interesting that, in the Gaussian space, marking can be worse than our deletion method in terms of distance computations.

5. Scalability

Finally, we show that the disk-based *EGNAT* scales well with the data size. We created synthetic datasets corresponding to 10-dimensional Gaussian vectors of size 10^5, 10^6, and 10^7, with the same distribution as before, and measured construction and search costs. We consider arity 19 for the *EGNAT*s. The results are given in Figure 9.

It can be seen that construction costs per element grow logarithmically with the data size in terms of distance evaluations and disk seeks and reads. That is, as the data size multiplies by 10, the curves shift upwards by a constant amount. The number of disk writes, instead, quickly stabilizes below the constant 2.

With respect to search costs, the traversed fraction of the database decreases as the database grows, exhibiting sharp sublinearity in query times both for distance evaluations and for disk reads/seeks. For example, a range query returning 1% of the database computes distances to more than 50% of the elements for $n = 10^5$, but just to 10% for $n = 10^6$, and to 5.5% for $n = 10^7$. The number of disk reads also grow sublinearily: as the data size multiplies by 10, the number of blocks read multiplies by 6. On more selective queries the sublinearity is even more pronounced.

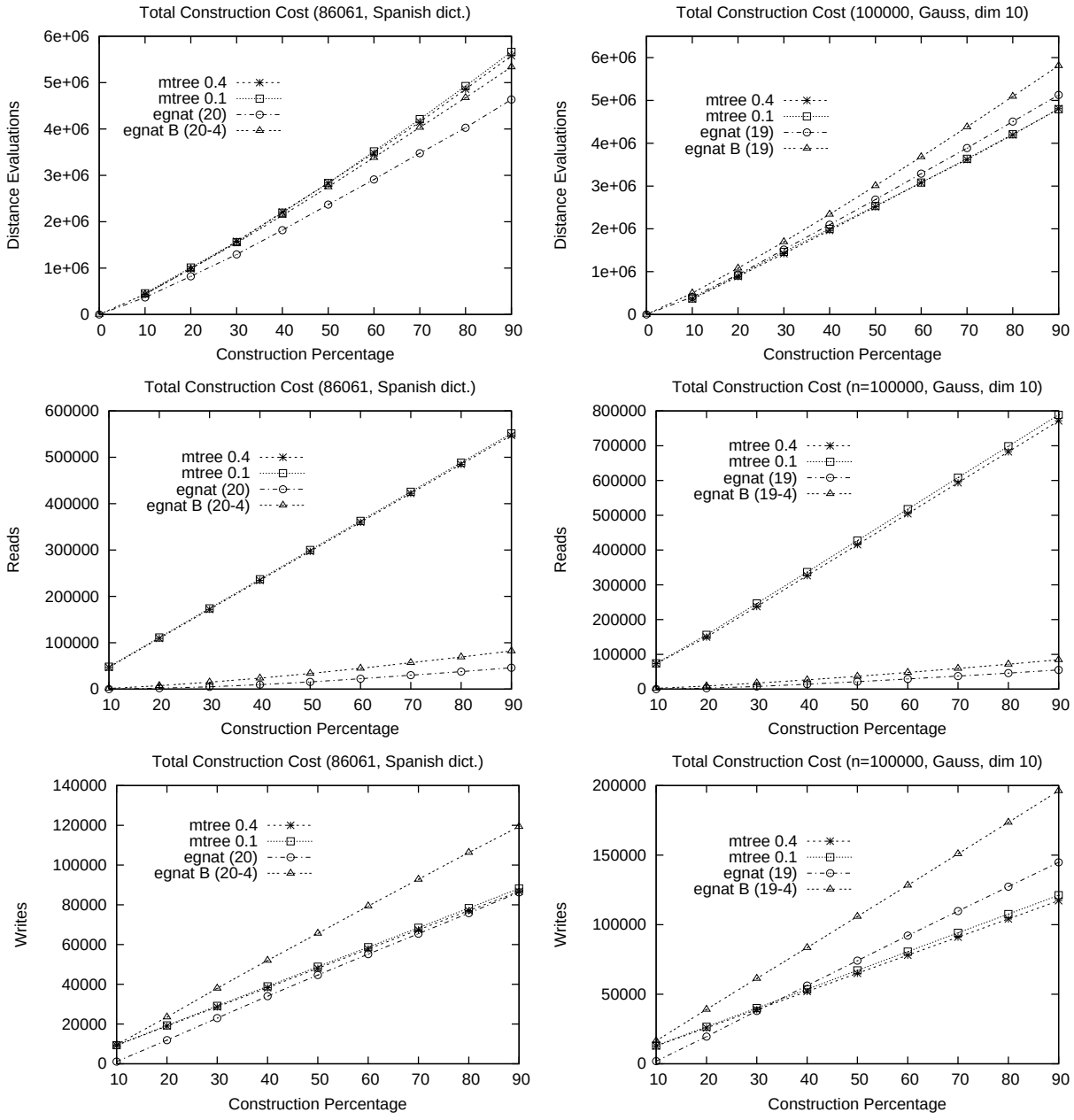

Figure 6: Construction costs of the secondary memory variants: distance computations, disk reads and writes.

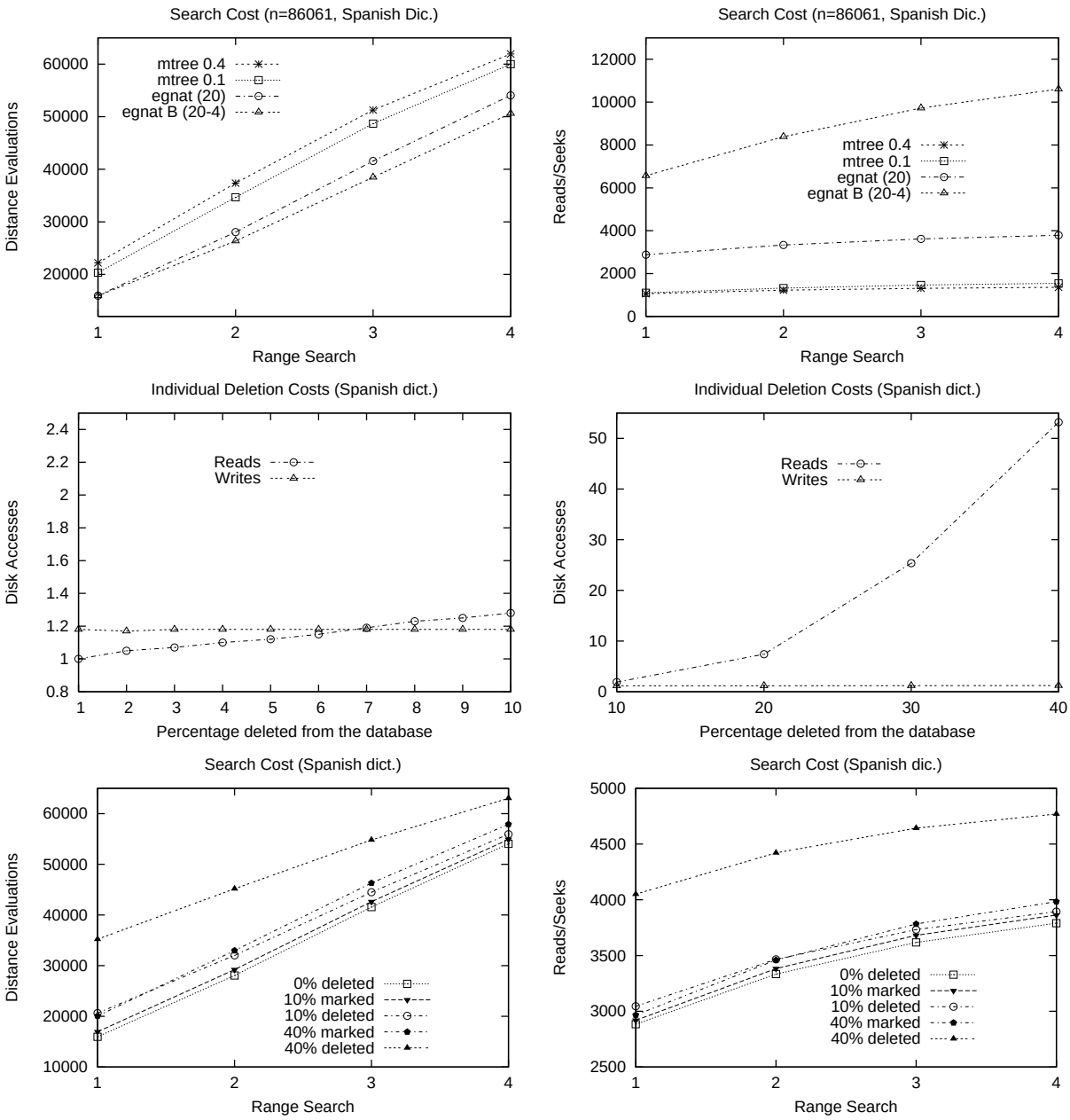

Figure 7: Costs of secondary memory variants on the dictionary words. Top: Search costs in distance computations and disk accesses. Middle: Deletion costs in I/Os, deleting 10% and 40% of the database. Bottom: Search costs after deletions, in distance computations and disk accesses.

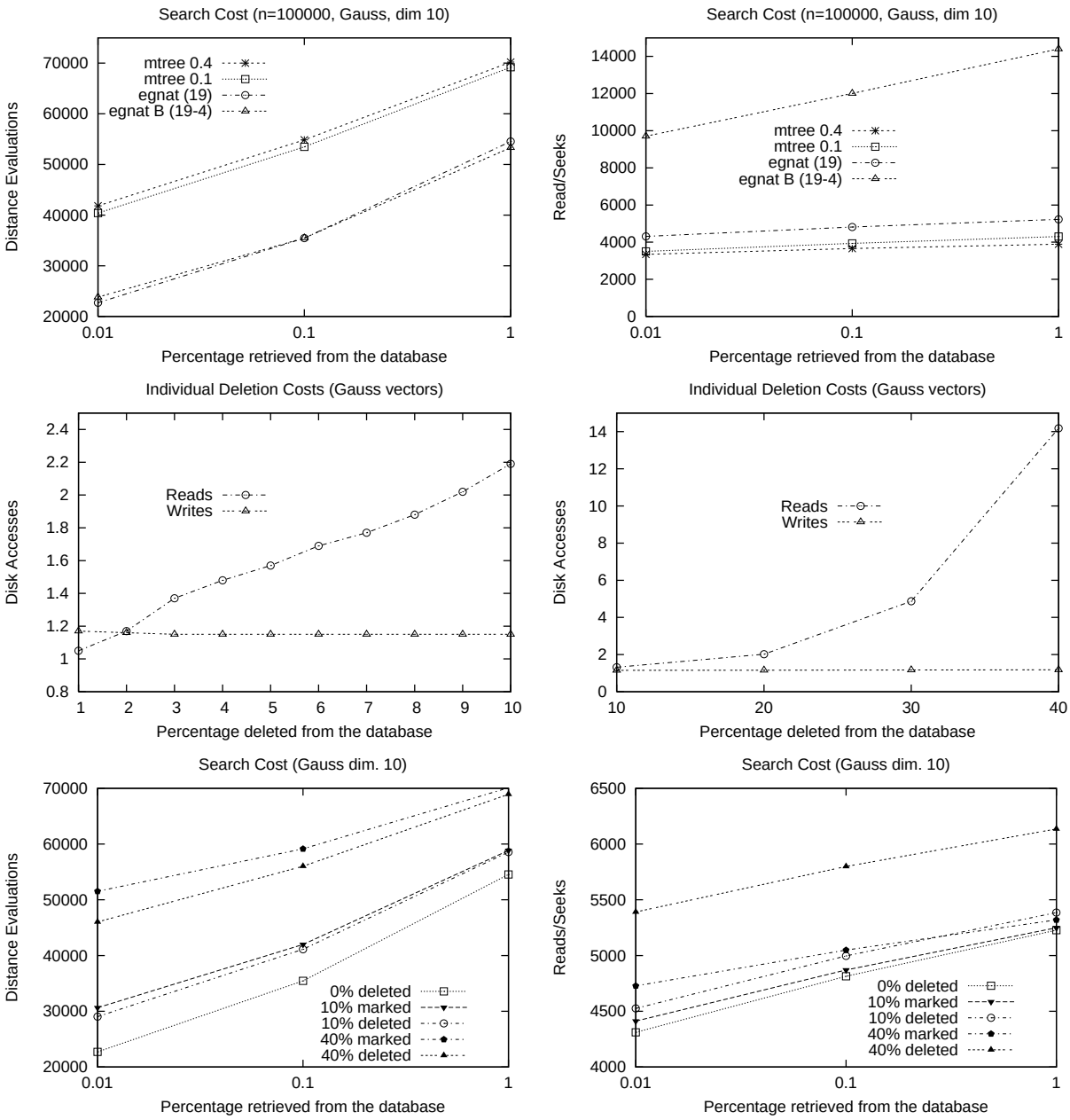

Figure 8: Costs of secondary memory variants on the Gaussian vectors. Top: Search costs in distance computations and disk accesses. Middle: Deletion costs in I/Os, deleting 10% and 40% of the database. Bottom: Search costs after deletions, in distance computations and disk accesses.

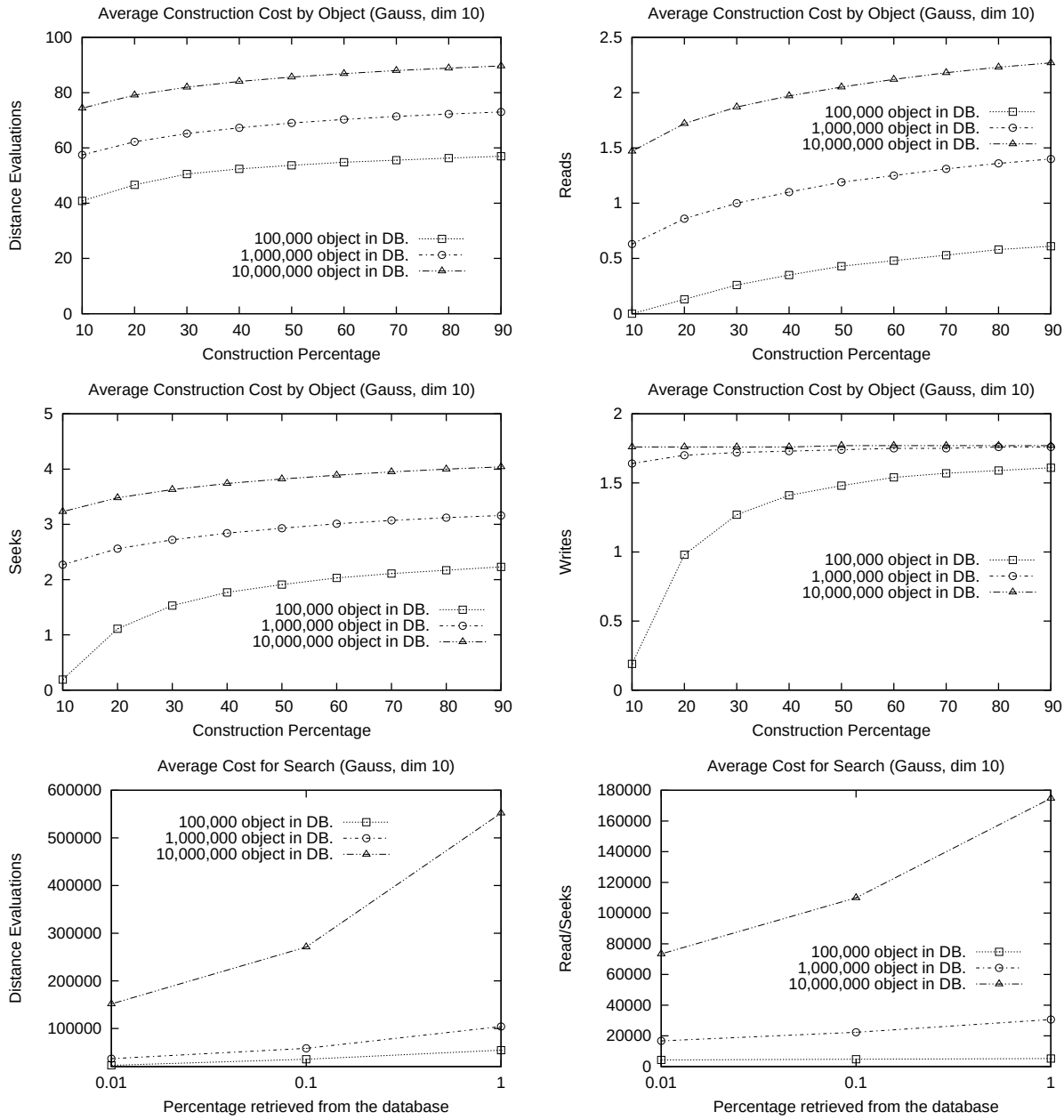

Figure 9: Scalability of *EGNATs* on larger datasets. The first four plots are construction cost per element in terms of distance computations, disk reads, seeks, and writes. The bottom row gives average search cost in terms of distance evaluations and disk reads/seeks.

6. Voronoi Trees

The concept of ghost hyperplanes is not only applicable to *GNATs*, but also to similar data structures. In this section we show how to apply the ghost hyperplane concept to a less known data structure, the *Voronoi tree (VT)* [3]. The *VT* is similar to the *Bisector Tree (BST [15])*, which in turn can be seen as a restriction of the *GNAT* structure, which only two objects per node are used and no $range_{i,j}$ information is stored, just the hyperplanes and the covering radii. That is, the *BST* is a binary tree where at each node 2 objects are chosen, each other point is sent to the subtree of its closer object, and subtrees are organized recursively.

The *VT* is an improvement over *BSTs*, where the tree has 2 or 3 objects (and children) per node. The insertion proceeds as in our *EGNAT*. However, when a new tree node has to be created to hold the inserted element, its closest element from the parent node is also inserted in the new node. *VTs* have the property that the covering radius is reduced as we move downwards in the tree, which provides better packing of elements in subtrees. It was shown [3] that *VTs* are superior and better balanced than *BSTs*. The search on a *VT* uses the covering radii to prune the search.

The remaining challenge on *VTs* is how to remove elements from them. Using ghost hyperplanes for deleting elements on a *VT* poses the challenge of the replicated elements: A single deletion of an object x may generate several ghost hyperplanes, as x is replicated in its subtree. However, note that all these positions form a single downward path from the highest occurrence of x towards a leaf. Therefore, we choose as a replacement an element y that shares the leaf node with the deepest occurrence of x. This way, we can use the same object y to replace all the occurrences of x, and we know that the choice is "good" for all these occurrences, as in all cases y descends from a leaf of the subtree of that occurrence of x.

Note that the property that the element replicated from the parent is the closest one may be lost after replacing x by y (or more precisely, remain valid only with tolerance $d(x, y)$). Yet, this property is not essential for the correctness of the search, which is carried out exactly as for the *EGNAT*. We also maintain the distance from each element to its closest parent, to further prune the search. This is also relaxed with the tolerance $d(x, y)$ if the parent x is replaced, just as for the *EGNAT*.

Figure 10 shows construction, deletion, and search costs for our VT, for the Gaussian space of dimension 10. We generalized the VT to arities larger than 3 (while maintaining the idea of copying the closer parent object to the child). The experimental results show that the lower arities, where the impact of copying the parent is higher, are indeed better. This is interesting, because more replication exists with lower arities, and nevertheless the search and deletion costs, after removing part of the database, are lower. Thus the better geometry of *VTs* is more relevant than the need of creating several ghost hyperplanes per deletion.

The comparison with the *EGNAT* is also interesting. On lower arities, the *VT* works better, thanks to its replication policy. On higher arities, where replication is still used but its impact is reduced, the *EGNAT* becomes superior. In particular, the *EGNAT* is a better alternative for secondary memory, where higher arities are necessary to make better use of the disk pages.

7. Conclusions and Future Work

Most metric search structures are static and do not handle secondary memory. The most famous exception is the *M-tree*, which is based on a ball partitioning of the space. This allows overlaps between regions, which gives flexibility to ensure good fill ratios and balancing of the tree, but in exchange has to traverse many branches at search time. In this paper we have presented the first, as far as we know, dynamic and secondary-memory-bound data structure based on hyperplane partitioning, the *EGNAT*. Hyperplanes do not have the flexibility of ball partitionings, but in exchange do not introduce overlapping areas.

Our experimental results show that, as expected, the *M-tree* achieves better disk page usage and consequently fewer I/O operations at search time, whereas our *EGNAT* data structure carries out fewer distance computations. On the other hand, construction costs are similar except that the *M-tree* requires many more disk page reads, presumably due to a more complex balancing policy.

Our main algorithmic contribution is a novel mechanism to handle deletions based on *ghost hyperplanes*, where the deleted element is replaced by another, and then overlapping is reintroduced as a way to avoid the expensive subtree reconstructions required in alternative hyperplane-based techniques [9]. We show that overlapping is robust enough to

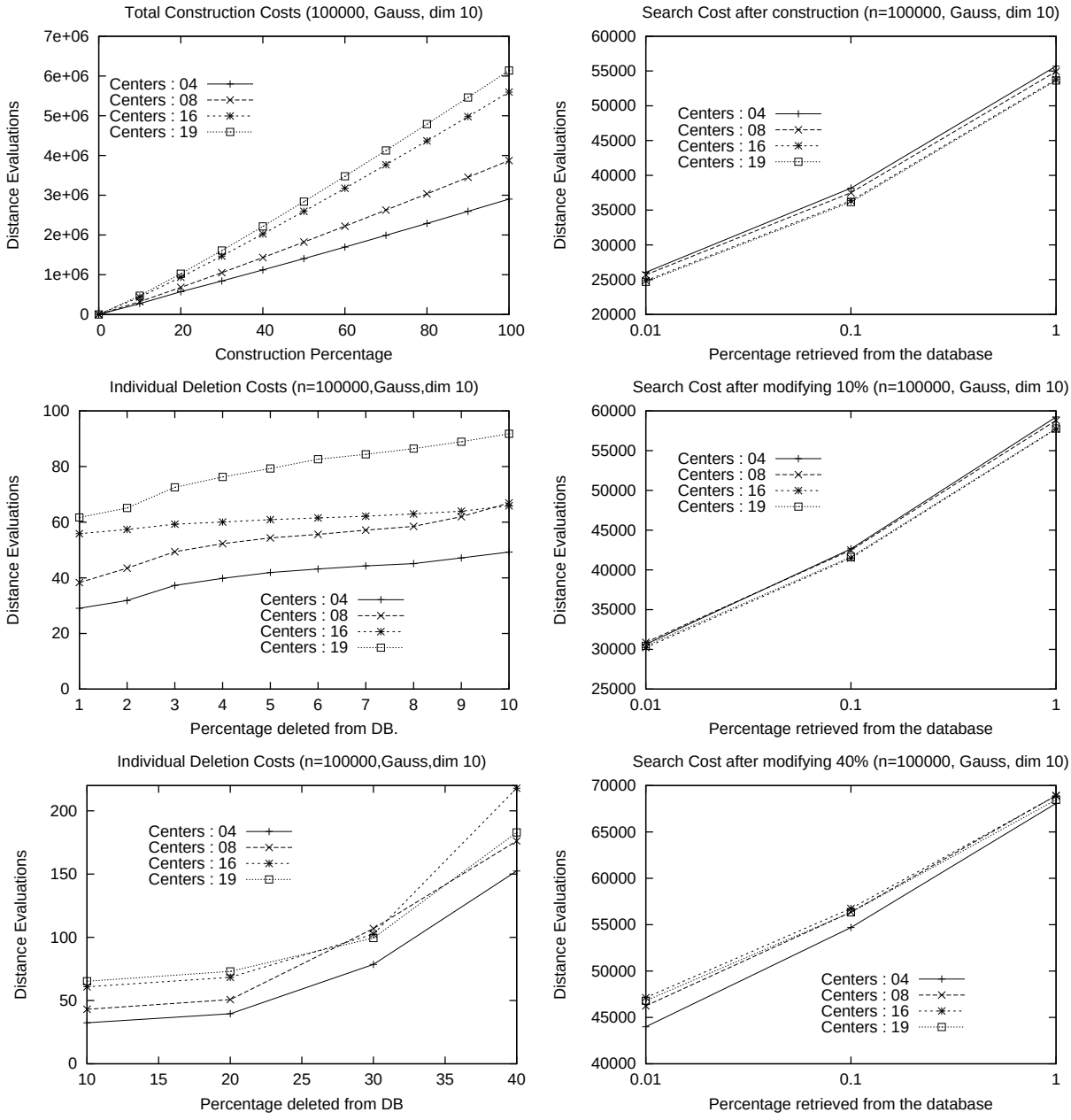

Figure 10: Construction, deletion, and search costs for *VTs*.

retain reasonable performance until a large fraction of the database has been deleted, and only then a reconstruction of subtrees from scratch would need to be carried out to restore good performance.

To illustrate the generality of the approach, we have applied ghost hyperplanes to *Voronoi trees* as well, obtaining a structure that works better than the *EGNAT* on low-arity trees. For secondary memory, where large arities are used, the *EGNAT* is still preferable. Ghost hyperplanes are likely to find applications in other hyperplane-based metric structures. For example, one could use them to support deletions in a recent proposal based on the *SAT* that achieves good space utilization [12].

In this paper we have assumed that the same metric index must be used to locate the element to be deleted, which triggers a zero-radius search in the structure that is responsible for most of the search cost. Indeed, this is a subproblem of different nature, which could be better solved for instance with secondary-memory hashing, taking care of keeping track of the disk position of each object identifier. This would make deletions much more efficient in terms of disk reads and distance computations.

Several lines of research remain open. For example, we could aim at smarter policies to choose the centers when a bucket overflows, which has been carefully studied, for example, for the *M-tree*. This could also be applied to the problem of improving disk page utilization. A more systematic study of subtree reconstructions is also necessary, to understand their impact in overall performance (for example, being more tolerant with overlaps involves fewer reconstructions but costlier operations, so the optimal amortized point is not clear).

Finally, it would be interesting to study how relaxation techniques similar to ghost hyperplanes can be used to achieve balanced hyperplane-partitioning trees, in particular disk-based ones. Balancing is a key to better space utilization and it could render the structure competitive with the *M-tree* in this aspect, while retaining the advantage of *EGNATs* in the others.

Acknowledgements

The first author has been partially funded by Fondecyt Projects 1-080019 and 1-1090037 (Chile). The second author has been partially funded by Universidad de Magallanes (PR-F1-02IC-08), Chile, and Universidad Nacional de la Patagonia Austral - UART (29/C035-1), Argentina. An early version of this paper appeared in *Proc. SISAP 2009*.

References

[1] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann, 2005.

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

[3] F. Dehne, H. Noltemeier, Voronoi trees and clustering problems, Informations Systems 12 (2) (1987) 171–175.

[4] S. Brin, Near neighbor search in large metric spaces., in: Proc. 21st Very Large Databases Conference (VLDB), 1995, pp. 574–584.

[5] P. Ciaccia, M. Patella, P. Zezula, M-tree : An efficient access method for similarity search in metric spaces., in: Proc. 23rd Very Large Databases Conference (VLDB), 1997, pp. 426–435.

[6] C. Traina, A. Traina, B. Seeger, C. Faloutsos, Slim-trees: High performance metric trees minimizing overlap between nodes, in: Proc. 7th Extending Database Technology (EDBT), 2000, pp. 51–61.

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

[8] J. Lokoc, T. Skopal, On reinsertions in M-tree, in: Proc. 1st Similarity Search and Applications (SISAP), IEEE CS, 2008, pp. 410–417.

[9] G. Navarro, N. Reyes, Dynamic spatial approximation trees, ACM Journal of Experimental Algorithmics 12 (2008) article 1.5.

[10] R. F. S. Filho, A. J. M. Traina, C. T. Jr., C. Faloutsos, Similarity search without tears: The OMNI family of all-purpose access methods, in: Proc. 17th International Conference on Data Engineering (ICDE), 2001, pp. 623–630.

[11] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, R. Zhang, idistance: An adaptive B+-tree based indexing method for nearest neighbor search, ACM Transactions on Database Systems 30 (2) (2005) 364–397.

[12] G. Navarro, N. Reyes, Dynamic spatial approximation trees for massive data, in: Proc. 2nd Similarity Search and Applications (SISAP), IEEE CS Press, 2009, pp. 81–88.

[13] E. Vidal, An algorithm for finding nearest neighbor in (approximately) constant average time, Pattern Recognition Letters 4 (1986) 145–157.

[14] M. Patella, Similarity search in multimedia databases, Ph.D. thesis, Dip. di Elect. Inf. e Sist., Univ. degli Studi di Bologna, Bologna, Italy (1999).

[15] I. Kalantari, G. McDonald, A data structure and an algorithm for the nearest point problem, IEEE Transactions on Software Engineering 9 (5).