EGNAT: A Fully Dynamic Metric Access Method for Secondary Memory

Roberto Uribe Paredes
Depto. de Ingeniera en Computación
Universidad de Magallanes
Punta Arenas, Chile
and Grupo de Bases de Datos - UART
Universidad Nacional de la Patagonia Austral
Río Gallegos, Argentina
roberto.uribe@umag.cl

Gonzalo Navarro
Dept. of Computer Science
University of Chile
Santiago, Chile
gnavarro@dcc.uchile.cl

***Abstract*—We introduce a novel metric space search data structure called *EGNAT*, which is fully dynamic and designed for secondary memory. The *EGNAT* is based on Brin's *GNAT* static index, and partitions the space according to hyperplanes. The *EGNAT* implements deletions using a novel technique dubbed *Ghost Hyperplanes*, which is of independent interest for other metric space indexes. We show experimentally that the *EGNAT* is competitive with the *M-tree*, the baseline for this scenario.**

I. 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 and focus on the simpler *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.

Metric space indexes are 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 [1].

Metric space data structures can be roughly divided into those based on so-called pivots and those based on clustering the space [1]. 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 the whole area by just comparing the query with its center. Indexes based on clustering are better suited for high-dimensional metric spaces, which is the most difficult problem in practice. Some clustering-based indexes are *GNAT* [2], M-Tree [3], *Slim-Tree* [4], and *SAT* [5].

There are two criteria to delimit areas in clustering-based structures, *hyperplanes* and *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*.

Most structures for metric spaces are not designed to support dynamism, that is, updates to the database once the index is built [1]. Yet, some structures allow non-massive insertion operations without degrading their performance too much. In recent years, indexes with full dynamic capabilities have appeared, such as *M-tree* [3], [6] and a dynamic *SAT* [7]. Deletions have proved to be more complex than insertions in Voronoi-type structures, especially to guarantee that the data structure quality is not degraded after massive updates [7].

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. Again, most of the structures for metric space search are designed for main memory. At present there are a few general methods for secondary memory. The best known is *M-tree*, and the *Slim-tree* [4] is based on it.

A problem of *M-tree* is that, because its clusters are based on covering radii, they overlap in space. Although this gives flexibility on where to insert new points, and is key to maintain the tree balanced, it also has a cost at search time: Even a query searched for exactly requires usually entering many subtrees. The *Slim-tree* is designed as a way to reduce those overlaps through periodic reorganizations, while there are dynamic methods as well [6].

The problem of overlapping branches does not exist in hyperplane partitioned data structures, such as *GNAT* or *SAT*. On the other hand, it is hard to maintain those trees balanced, as in principle there is no control over the insertion positions of new elements. Deletions are also much more difficult, as there is no simple way of replacing a deleted center with another while ensuring that all the existing subtrees are covered by the correct areas (this can be achieved with *M-tree* by enlarging covering radii).

This difficulty to choose the locations of inserted objects makes also difficult 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 designing *EGNAT*, a dynamic variant of *GNAT* data structure [2]. We introduce novel techniques to deal with the problems of the rigid structure, so as to maintain the data structure quality upon insertions and deletions, while keeping their cost low and the space utilization high. The result is shown experimentally to be superior to *M-tree* in various scenarios, thanks to the intrinsic advantage given by the nonoverlapping areas.

II. *EGNAT*

As explained, the *EGNAT* is based on the concept of Voronoi Diagram. It is a variant of the *GNAT* index proposed by Brin [2]. The *GNAT* is a k-ary tree built by selecting k centers (database objects) at random, $\{p_1, p_2, \dots, p_k\}$, 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 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}, $range_{i,j} = (\min\{d(p_i, x), x \in D_{p_j}\}, \max\{d(p_i, x), x \in D_{p_j}\})$.

The *EGNAT* contains two types of nodes, *buckets* (leaves) and *gnats* (internal nodes). Nodes are initially created as *buckets*, maintaining only the distances from each point to the parent element. This yields a significant reduction in space, while still allowing some pruning by applying the triangle inequality using the distance to the parent. 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. However, 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.

[1]The real *GNAT* algorithm is more complex, as it avoids comparing the query with some centers, by using an AESA-like [8] algorithm over the zones. The *EGNAT* uses the simpler method we present here.

Algorithm 1 *gnat*: search with range r for query q.

```
rangesearch(Node P, Query q, Range r)
 1: $c \leftarrow \text{argmin}_{1 \le i \le k} d(p_i, q)$
 2: $d_{min} \leftarrow d(p_c, q)$
 3: for all $p_i \in P$ do
 4:    if $[d_{min} - r, d_{min} + r] \cap range_{i,c} \neq \emptyset$ then
 5:       if $d(p_i, q) \le r$ then
 6:          Report $p_i$
 7:       end if
 8:       rangesearch($D_{p_i}, q, r$)
 9:    end if
10: end for
```

III. Deleting in the *EGNAT*

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.

We discard the alternatives of full subtree reconstruction for being too expensive, and that of retaining the nodes and just marking them as deleted, as in usual metric space applications objects are too large and we cannot afford maintaining deleted objects just for the sake of the index.

In this section we describe and evaluate an innovative solution to this problem, called *ghost hyperplanes*.

We note, before entering into those 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 ($O(km)$ distance computations) for converting a bucket of size m into a *gnat* node. Periodic subtree reconstructions handle this problem.

A. *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 holded x is then marked as *affected*, and the distance between x and y is recorded in the node.

As there is no re-calculation of $range_{i,j}$, the partition virtually changes its shape, creating an overlap with other areas. This has to be accounted for at the time of searching, insertion, and deletion in the *EGNAT*. This method is called *ghosts hyperplanes* because two hyperplanes (the old and the new) coexist. Fig. 1 illustrates.

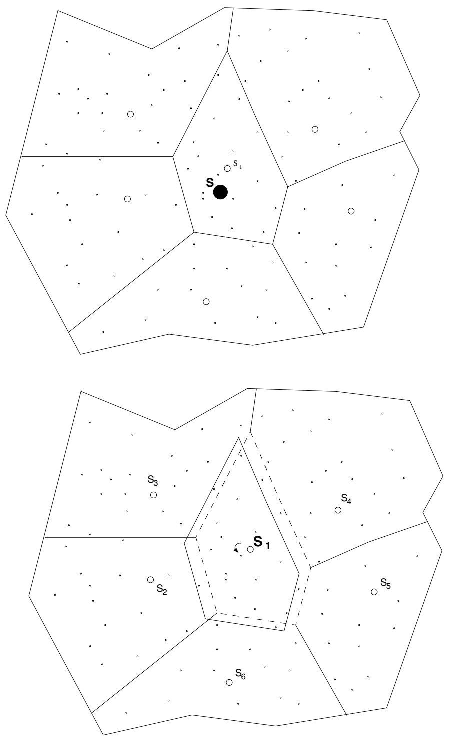

Figure 1. Partition of the space before and after a deletion.

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 and the replaced element. Then, if we consider the new element and still want to find those inserted before the replacement, we must consider that the hyperplanes may be off up to I.

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. Note also that 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 nodes will maintain an I value ($I = 0$ when no deletion has occurred), and upon each replacement I will be increased by the distance between the original and replaced elements. We might rebuild a subtree when its I values have reached a point that queries become too inefficient.

B. 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 nearest neighbor needs not be in a leaf, moving it to replace the deleted object triggers a new deletion subproblem. Although there is no guarantee of termination, if one repeats the process there is a high chance of finishing after very few iterations, as most of the elements are in leaves. Yet, a single deletion introduces several ghost hyperplanes, which may be worse than making just one replacement with a not-so-close element in the first place.

2. The nearest descendant: A way to guarantee termination is to choose the nearest neighbor from the descendants of the node to be deleted. This limited nearest neighbor search is likely to produce a good candidate, as we search within the partition of the query element.

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 leaf of the bucket is chosen.

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

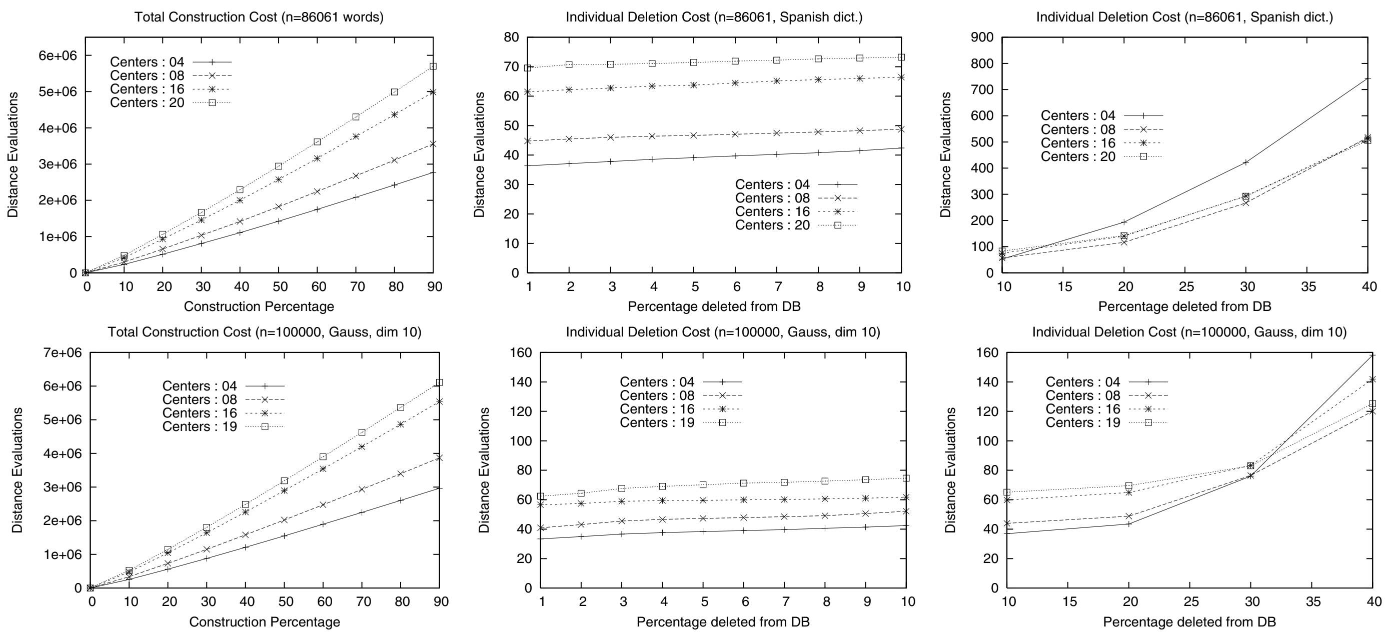

Figure 2. Aggregated construction costs (left) and individual deletion costs when deleting the first 10% (middle) and 40% (right) of the database. The different lines refer to different tree arities (centers per node).

C. Experimental Evaluation

Tests made in two metric 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 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 Fig. 2 (left), 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 shown in Fig. 2. 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%, right), 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.

Fig. 3 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, in most cases, the deleted elements are at leaves, where the policies do not play any role.

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.

Fig. 4 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

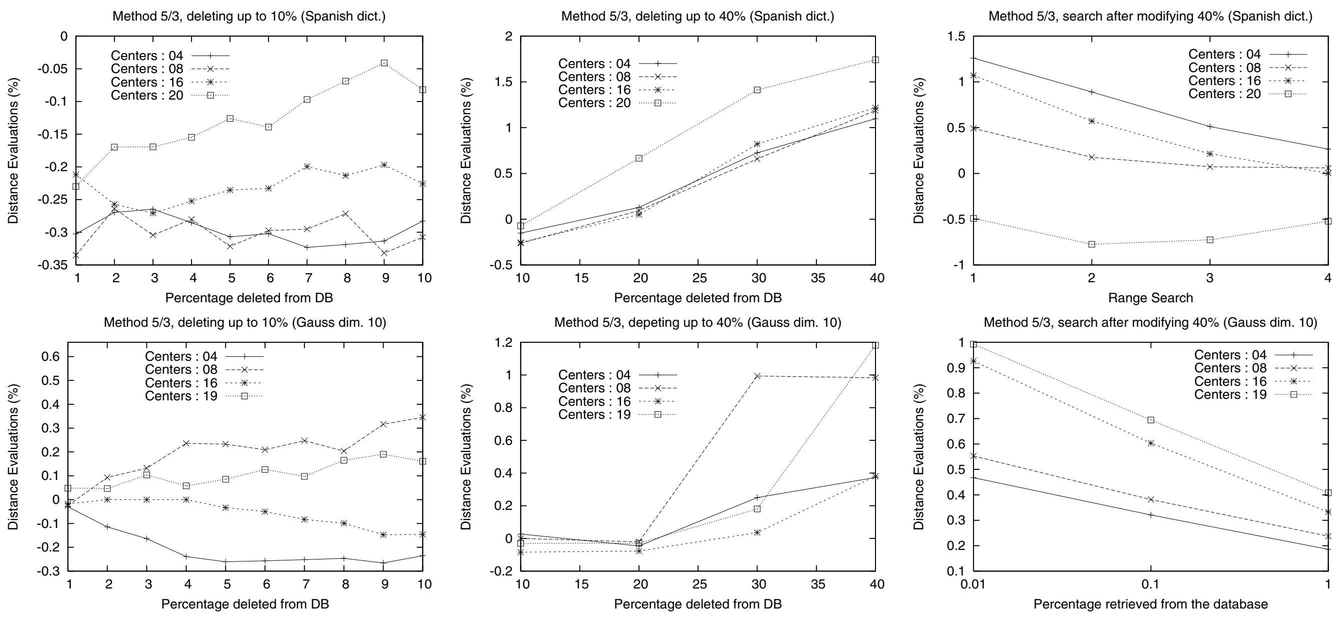

Figure 3. Ratio between methods, in deletion (left, middle) and search (right) costs.

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.

IV. Secondary Memory

In secondary memory we have to 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 search in secondary memory: the *M-tree* [3]. Deletions are not implemented in the standard *M-tree* distribution we use for comparison[2], 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 [9].

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. By sharing, for example, disk pages among 6 buckets, the disk page utilization raises to 25% on both spaces. Sharing among 4 buckets, utilization is 22%.

Aggregated construction costs can be seen in Fig. 5. 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* carries out many more disk block reads. The number of disk writes, instead, grows slower on the *M-tree*.

Fig. 6 (left column) compares search costs, in terms of distance evaluations and number of reads or seeks (the number is the same for both). *EGNAT-B* refers to the *EGNAT* sharing pages among 4 buckets. It can be seen that the *EGNAT* performs (sometimes significantly) fewer distance computations, 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.

Fig. 6 (middle column) shows 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 with the percentage of the database deleted (this makes sense, as the process of finding the element to delete is read-only).

Finally, Fig. 6 (right column) shows the search costs, in terms of distance computations and disk accesses, after a portion of the database is deleted and reinserted. We

[2]`http://www-db.deis.unibo.it/Mtree`

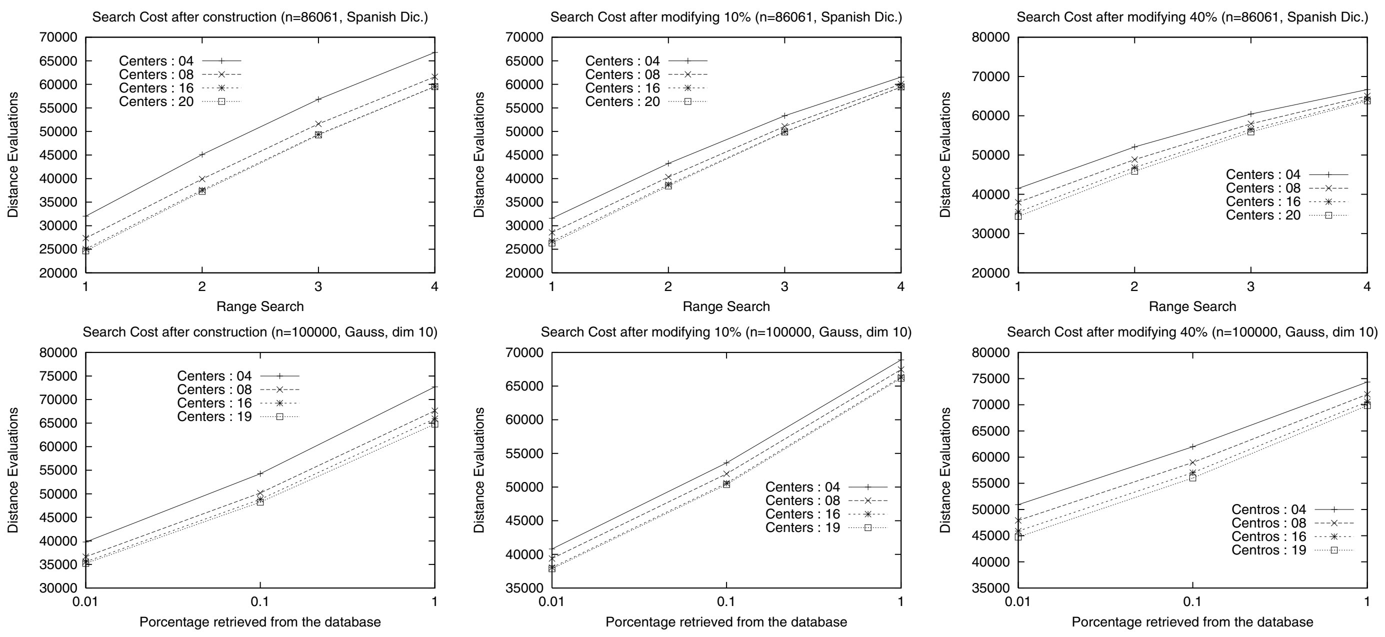

Figure 4. Search costs after construction and after deleting and reinserting part of the database.

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, this can be worse than our deletion method in terms of distance computations.

V. 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, *EGNAT*. This does not have the flexibility of ball partitionings, but in exchange does 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 (which is not present in the *M-tree*) based on *ghost hyperplanes*, where the deleted element is replaced by another, and then overlapping is reintroduced as a way to avoid expensive tree reconstructions advocated in alternative hyperplane-based techniques [7]. We show that overlapping is robust enough to retain reasonable performance until a large fraction of the database has been deleted, and only then a reconstruction of subtrees from scratch would be carried out to restore good performance.

We have assumed that the same *EGNAT* structure 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, ghost hyperplanes are applicable to other similar data structures. In particular, we have applied them successfully to the *Voronoi tree* [10], not shown here for lack of

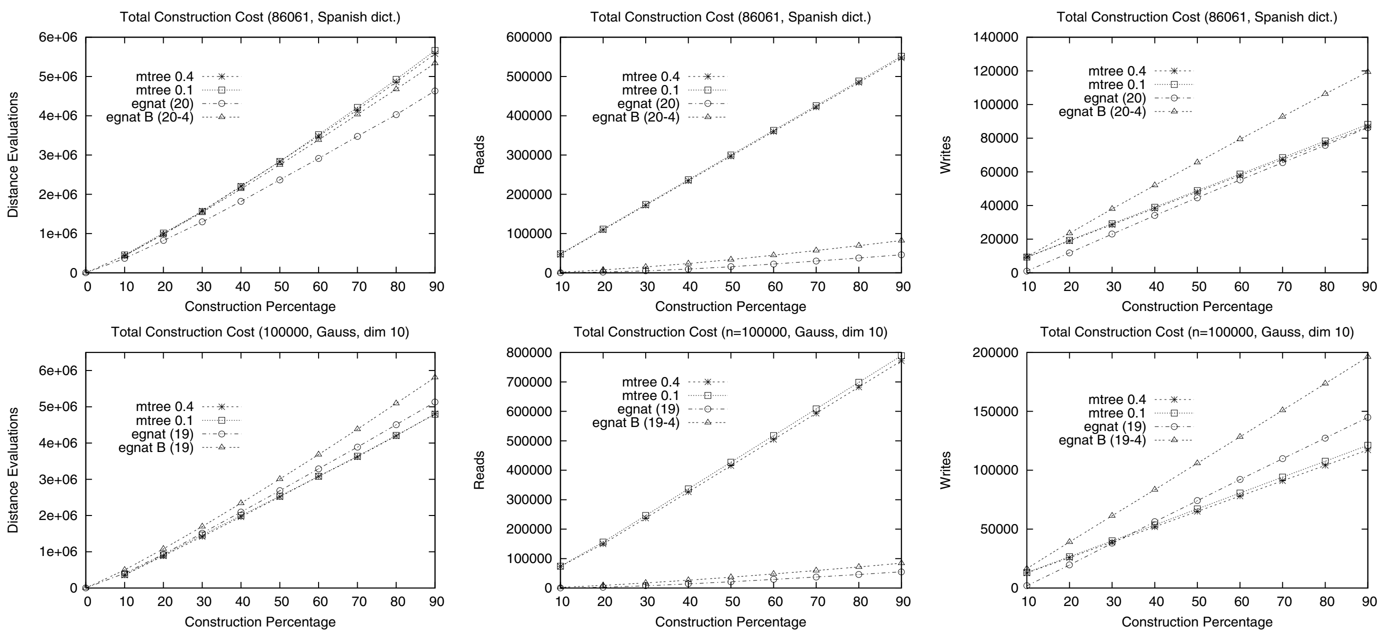

Figure 5. Construction costs of the secondary memory variants: distance computations, disk reads and writes.

space. Ghost hyperplanes have also been applied to the *SAT* [7], where they were shown to be extremely competitive.

Acknowledgements

The first 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. The second author has been partially funded by Fondecyt Project 1-080019 (Chile).

References

[1] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín, "Searching in metric spaces," *ACM Computing Surveys*, vol. 33, no. 3, pp. 273–321, 2001.

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

[3] P. Ciaccia, M. Patella, and 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.

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

[5] G. Navarro, "Searching in metric spaces by spatial approximation," *The Very Large Databases Journal*, vol. 11, no. 1, pp. 28–46, 2002.

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

[7] G. Navarro and N. Reyes, "Dynamic spatial approximation trees," *ACM Journal of Experimental Algorithmics*, vol. 12, p. article 1.5, 2008.

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

[9] M. Patella, "Similarity search in multimedia databases," Ph.D. dissertation, Dip. di Elect. Inf. e Sist., Univ. degli Studi di Bologna, Bologna, Italy, 1999.

[10] F. Dehne and H. Noltemeier, "Voronoi trees and clustering problems," *Informations Systems*, vol. 12, no. 2, pp. 171–175, 1987.

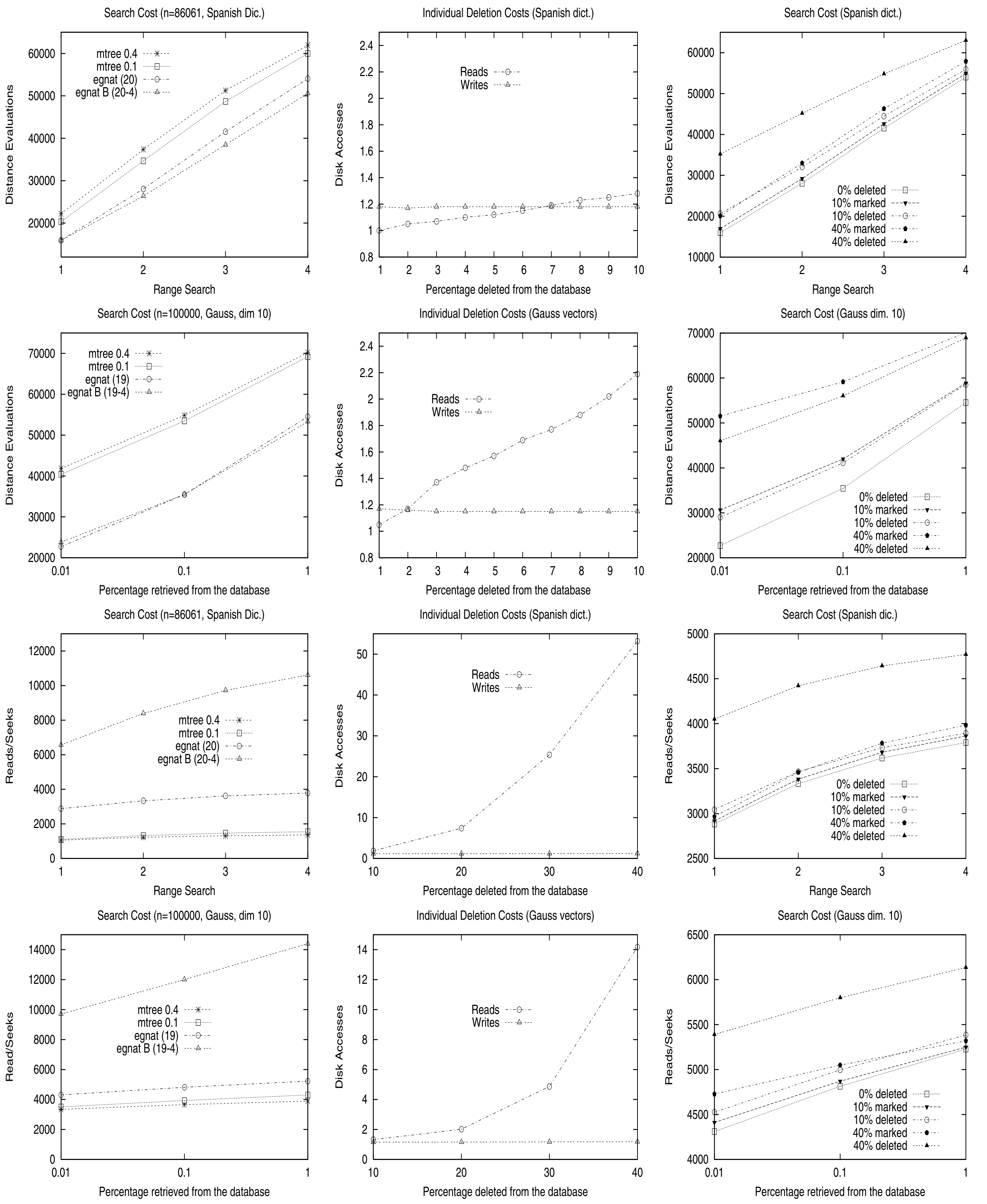

Figure 6. Costs of secondary memory variants. Left column: Search costs in distance computations and disk accesses. Middle column: Deletion costs in I/Os, deleting 10% and 40% of the database. Right column: Search costs after deletions, in distance computations and disk accesses.