Dynamic Spatial Approximation Trees for Massive Data

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

Nora Reyes
Depto. de Informática
Universidad Nacional de San Luis
San Luis, Argentina
nreyes@unsl.edu.ar

***Abstract*—Metric space searching is an emerging technique to address the problem of efficient similarity searching in many applications, including multimedia databases and other repositories handling complex objects. Although promising, the metric space approach is still immature in several aspects that are well established in traditional databases. In particular, most indexing schemes are not dynamic, that is, few of them tolerate insertion of elements at reasonable cost over an existing index and only a few work efficiently in secondary memory.**

In this paper we introduce a secondary-memory variant of the Dynamic Spatial Approximation Tree, which has shown to be competitive in main memory. The resulting index handles well the secondary memory scenario and is competitive with the state of the art, becoming a useful alternative in a wide range of database applications. Moreover, our ideas are applicable to other secondary-memory trees where there is little control over the tree shape.

I. Introduction

Similarity searching [1], [2] has applications in many fields, such as multimedia databases, text retrieval, function prediction, and many others. All those applications share some common characteristics. There is a finite *dataset* of elements belonging to a *metric space*, where a *distance function* is used to assess similarity. *Similarity queries* are posed to this dataset. These consist basically in, given a new element of the space called the *query*, looking for elements of the dataset that are similar enough to the query. The dataset is preprocessed so as to build an *index* that reduces query time. This metric space approach is becoming widely popular [1], [2] and a large number of indexing methods have flourished [1], [3], [4], but mature solutions from the database viewpoint are a long way off.

Most of the existing indexes are *static*: Once they are built for a given dataset, adding more elements to the dataset, or removing an element from it, requires an expensive updating of the index. Some indexes tolerate insertions in principle, but their quality degrades and require periodic rebuildings. Others tolerate deletions with the same quality degradation problem. Thus there are very few *dynamic* indexes.

There are also many interesting databases for similarity searching where the objects are so large that they must stay on disk; or the objects are so many that the index itself cannot fit in main memory. In this case, although the similarity computation can be expensive (e.g., taking milliseconds of CPU time) we cannot disregard disk costs.

From the few dynamic indexes, even fewer work well in secondary memory. That is, most of them need the data structure in main memory in order to operate efficiently.

Although for some applications a static scheme may be acceptable, many relevant ones do require dynamic capabilities. Actually, in many cases it is sufficient to support insertions, such as in digital libraries and archival systems, versioned and historical databases, and several other scenarios where objects are never updated or deleted.

In this paper we introduce a dynamic index aimed at secondary memory. We base our work on the Dynamic Spatial Approximation Tree (*dsa-tree*) [5]. It has been shown that the *dsa-tree* gives an attractive tradeoff between memory usage, construction time, and search performance. Our secondary memory versions (*dsa*-tree* and *dsa+-tree*) retain these good features, and in addition perform well in secondary memory. We focus on handling insertions (and thus incremental construction) and searches in this paper, leaving deletions for future work. Our experimental comparisons show that our structures achieve very good disk page utilization and are competitive with the *M-tree* (the best known dynamic secondary-memory metric index), being more efficient for insertions and comparable for searches.

The *dsa*-tree* ensures a minimum fill ratio of 50% and achieves 75%–85% in practice. Instead, the *dsa+-tree* is a heuristic that does not ensure a minimum fill ratio, achieving 65%–70% in experiments. In terms of I/Os, the *dsa+-tree* is costlier to build but faster to search than the *dsa*-tree*.

Although we will focus on range searches in this work, the structures are capable of nearest neighbor searching in a range optimal-way, by simply inheriting the corresponding main-memory algorithms [5], [6].

II. 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 satisfies the three axioms that make $(\mathbb{U}, d)$ a *metric space*: strict positiveness, symmetry, and triangle inequality. The smaller the distance between two objects, the more "similar" they are. 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. We focus on the *range query* in this work: Given $q \in U$ and $r > 0$, retrieve all elements of S within distance r to q. That is, $\{x \in S\ ,\ d(x,q) \leq r\}$.

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

III. Dynamic Spatial Approximation Trees

In this section we recall the Dynamic Spatial Approximation Tree (*dsa-tree*) [5], which inherits from its static variant [7]. Unlike most other structures, based on dividing the search space, the *dsa-tree* is based on the idea of approaching the query spatially, that is, starting at the tree root and getting closer and closer to the query, with a tolerance given by the query radius.

A number of alternatives for insertion of new elements into a *dsa-tree* have been discussed and evaluated [5]. In this section we describe only the best one, which we have used in our *dsa*-tree* and *dsa+-tree*. This is a combination of *timestamping* and *bounded arity*. A maximum tree arity $MaxArity$ (maximum number of children per node) is fixed, and also a timestamp of the insertion time of each element is kept. Each tree node a maintains its neighbor set $N(a)$, its timestamp $T(a)$, and its covering radius $R(a)$ (maximum distance to a subtree element, used to prune the searches). Whenever a new inserted element x arrives at node a, we check whether it is closer to a than to any element in $N(a)$. If so, 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 each element is older than its children and than its next sibling. See Algorithm 1.

In general, the idea of the tree is that elements should be 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. This permits that the search looks for the closest element to the query q in $N(a)$, yet with tolerance given by the search radius r. However, when a new element is inserted into $N(a)$, other elements that were inserted into 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.

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

```
Insert(Node a, Element x)

1. R(a) ← max(R(a), d(a,x))
2. c ← argmin_{b∈N(a)} d(b,x)
3. If d(a,x) < d(c,x) ∧ |N(a)| < MaxArity Then
4.    N(a) ← N(a) : x
5.    N(x) ← ⟨⟩,  R(x) ← 0
6.    T(x) ← CurrentTime
7.    CurrentTime ← CurrentTime + 1
8. Else Insert(c, x)
```

In principle, at search time 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 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 depicts the search process. Note that $d(a,q)$ is always known except in the first invocation, and the initial t is $+\infty$.

IV. Secondary Memory

Our secondary-memory implementation maintains exactly the same structure and carries out the same distance evaluations of the main-memory version. The challenge is how to maintain a disk layout in order to minimize I/Os. We describe the *dsa*-tree* in this section, and will spot the differences with the *dsa+-tree* in the next.

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

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

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

1. If T(a) < t  ∧  d(a,q) ≤ R(a) + r Then
2.    If d(a,q) ≤ r Then Report a
3.    d_min ← ∞
4.    For b_i ∈ N(a) Do // ascend. timestamps
5.       If d(b_i,q) ≤ d_min + 2r Then
6.          t' ← min{t} ∪ {T(b_j), j > i ∧
                          d(b_i,q) > d(b_j,q) + 2r}
7.          RangeSearch(b_i, q, r, t')
8.          d_min ← min{d_min, d(b_i,q)}
```

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.

A. 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 point to another page) have two parts: the page and the node offset inside that page. Fig. 1 illustrates the $F(a)$ and $S(a)$ pointers (while $T(a)$ 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.

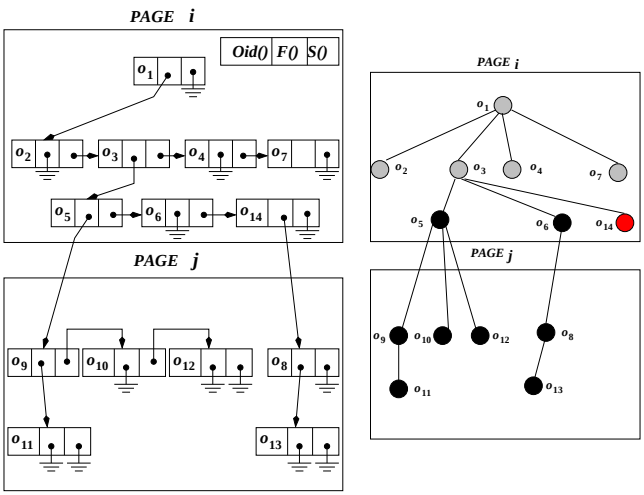


Figure 1. Example of $F(a)$ and $S(a)$ pointer layout.

B. Insertions

To insert a new element x into the *dsa*-tree* 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. Else, 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 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.

C. 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 Fig. 2.

2nd (vertical split) If the page of $N(a)$ contains subtrees with different parents from another page, we make room by

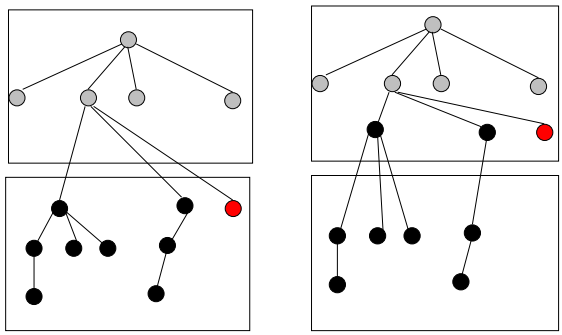


Figure 2. Example before (left) and after (right) applying the policy ***move to parent***.

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 $N(a)$. Fig. 3 illustrates this case.

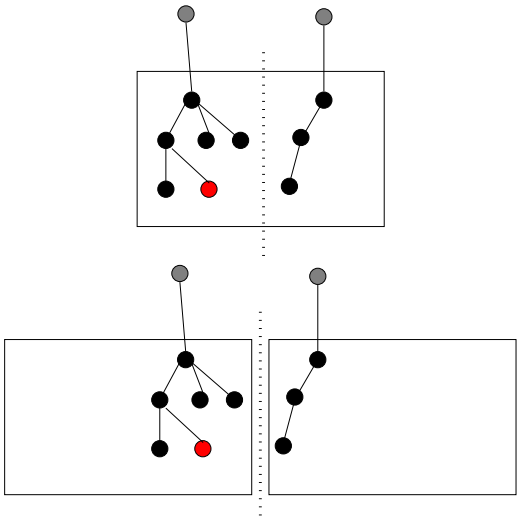


Figure 3. Example before (top) and after (bottom) applying the policy ***vertical split***.

To maintain a property whose purpose will be apparent in Section IV-D, 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. 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 a $N()$ 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.

Fig. 4 illustrates a horizontal split.

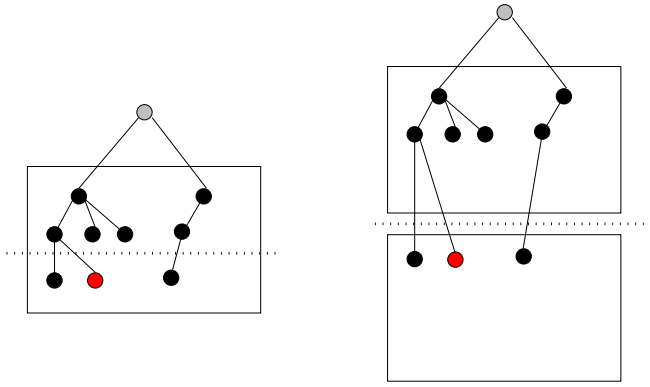


Figure 4. Example before (left) and after (right) applying the policy ***horizontal split***.

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.

D. 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).

E. Searches

Searches proceed exactly as in the *dsa-tree*, 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 [7].

V. A Heuristic Variant

The *dsa*-tree* 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, *dsa+-tree*, 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 *dsa*-tree* are as follows. In the *dsa+-tree* 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 "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.

VI. Experimental Results

In order to give a broad picture of the performance of our index, we have selected four widely different metric spaces, all from the SISAP Metric Library (`www.sisap.org`).

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.

Documents: 1,265 documents under the Cosine similarity, from TREC-3 collection. In this model the space has one coordinate per term and documents are seen as vectors in this space. The distance we use is the angle among the vectors.

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

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

For the search experiments, we built the indexes 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 have considered range queries retrieving on average 0.01%, 0.1% and 1% of the dataset. This corresponds to radii 0.140000, 0.150000 and 0.195000 for the documents, 0.605740, 0.780000 and 1.009000 for the images, and 0.051768, 0.082514 and 0.131163 for the histograms. Words have a discrete distance, so we used radii 1 to 4, which retrieved on average 0.00003%, 0.00037%, 0.00326% and 0.01757% of the dataset, respectively. The same queries were used for all the experiments on the same datasets. Given the existence of range-optimal algorithms for k-NN searching [6], we have not considered these search experiments separately, as their search cost is exactly that of range searching with a radius that captures the k neighbors.

The *M-tree* [8] is the best-known dynamic and secondary-memory data structure, and its code is freely available[1]. We have used the parameter setting suggested by the authors[2]. Another relevant alternative is the *D-index*, which has been shown to perform similarly to the *M-tree* [9], [10]. Thus we only compare to the *M-tree* in this conference version.

The disk page size is 4KB. Both data structures cache the tree root in main memory.

A. Construction

We build the *dsa*-tree* and *dsa+-tree* by successive insertions, using the $MaxArity$ that performed best for the main-memory setup [5]. This is 4 for all the spaces but words, where it is 32. We tried other arities for the secondary-memory scenario, but these arities are still the best.

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

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

Table I shows the average disk page occupancy achieved for the different spaces. As explained, this is guaranteed to be at least $1/2$ for the *dsa*-tree*, but in practice it is $3/4$ to $5/6$. That of the *dsa+-tree* is around $2/3$, which coincides with typical B-tree disk page occupancies ($1/\ln 2 \approx 69\%$). We also show the total number of disk pages used, compared to the *M-tree*. The *dsa*-tree* uses (usually much) less space than the *dsa+-tree*, and this in turn uses significantly less than the *M-tree*.

Dataset	Fill ratio		Total pages used		
	dsa-tree*	*dsa+-tree*	*dsa*-tree*	*dsa+-tree*	*M-tree*
Words	83%	66%	904	1,536	1,608
Documents	84%	68%	12	22	31
Images	80%	67%	1,271	1,726	1,973
Histograms	75%	67%	18,781	21,136	31,791

Table I
AVERAGE SPACE USAGE FOR THE DIFFERENT DATASETS.

Construction costs are shown in Figs. 5 (distances) and 6 (I/Os). In both aspects, the *dsa*-tree* and *dsa+-tree* build significantly faster than the *M-tree*: 25%–80% of the distance computations and 35%–80% of the I/Os. The *dsa*-tree* uses (sometimes significantly) fewer I/Os than the *dsa+-tree*, as it does not modify pointers in several disk pages. In practice the insertion cost on the *dsa*-tree* and *dsa+-tree* grows very slowly (proportionally to the tree depth): up to 25–36 distance computations (except ≈ 80 for words) and up to 2.0–5.5 I/Os (except around 10.5 for histograms).

B. Searches

Search costs for the *dsa*-tree*, *dsa+-tree*, and *M-tree*, are shown in Figs. 7 (distance evaluations) and 8 (pages read). Let us call collectively *dsa-tree* the first two.

The result of the comparison is mixed. In documents, the *M-tree* is better in both aspects, where in histograms the *dsa-tree* is much better in both aspects. In words, the *M-tree* is much better in I/Os[3] but the *dsa-tree* is better in distance computations. Finally, on images the *dsa-tree* is better in distance computations, and the outcome in I/Os depends on the search radii; the *M-tree* is better for larger radii. Note that the *dsa*-tree* uses the same number of distance computations, but (sometimes many) more I/Os, than the *dsa+-tree*. This shows that the consequences of a better packing are a poorer search performance in I/Os.

VII. CONCLUSIONS

We have presented a secondary-memory variant of the Dynamic Spatial Approximation Tree [5], which retains the original tree structure (and hence identical search and construction costs in terms of distance evaluations). Thus the focus is on disk page layout policies that achieve competitive performance in terms of I/Os. We have shown that the resulting structure achieves good space utilization ($2/3$ to $5/6$) and is competitive in distance computations and I/Os: It is consistently better than the *M-tree* for insertions, and better or worse for searches, depending on the metric space.

Our techniques are general, and could be useful in other scenarios where there is little control over the shape of the tree when insertions are carried out.

The most important remaining work is to handle deletions. Several such policies have already been studied for the main-memory variant [5], thus the focus will be, again, in achieving secondary-memory versions that retain good space utilization and are I/O-efficient. Designing bulk-loading mechanisms is also an interesting problem: In main memory, the construction by succesive insertions worked better than the static construction [5]. Finally, alternative page allocation policies might improve I/Os, e.g. trying to share pages among "close" subtrees.

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

ACKNOWLEDGEMENTS

Partially funded by Fondecyt Grant 1-080019, Chile (both authors) and by the Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile (first author).

REFERENCES

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

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

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

[4] G. Hjaltason and H. Samet, "Index-driven similarity search in metric spaces," *ACM TODS*, vol. 28, no. 4, pp. 517–580, 2003.

[5] G. Navarro and N. Reyes, "Dynamic spatial approximation trees," *ACM JEA*, vol. 12, p. article 1.5, 2009, 68 pages.

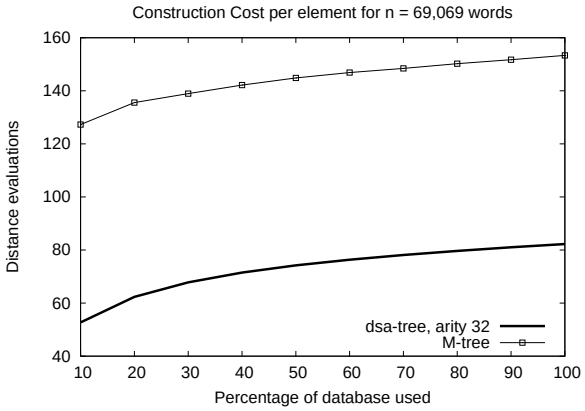
[6] G. Hjaltason and H. Samet, "Incremental similarity search in multimedia databases," Univ. of Maryland, Comp. Sci. Dept., Tech. Rep. CS-TR-4199, 2000.

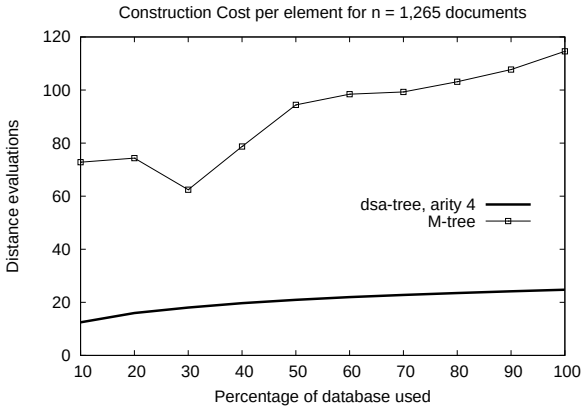
[7] G. Navarro, "Searching in metric spaces by spatial approximation," *The VLDB Journal*, vol. 11, no. 1, pp. 28–46, 2002.

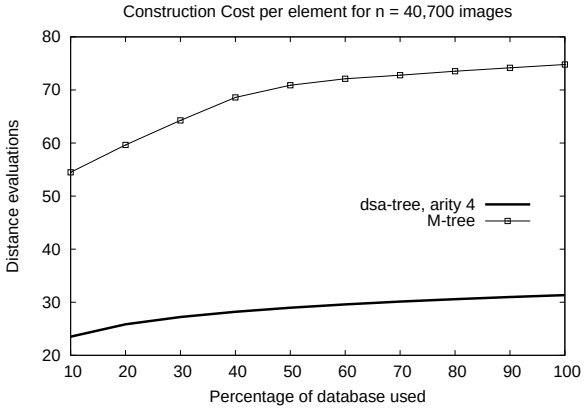
[8] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: an efficient access method for similarity search in metric spaces," in *Proc. 23rd VLDB*, 1997, pp. 426–435.

[9] V. Dohnal, "An access structure for similarity search in metric spaces," in *EDBT Workshops*, 2004, pp. 133–143.

[10] V. Dohnal, C. Gennaro, P. Savino, and P. Zezula, "D-index: Distance searching index for metric data sets," *MTAP*, vol. 21, no. 1, pp. 9–33, 2003.







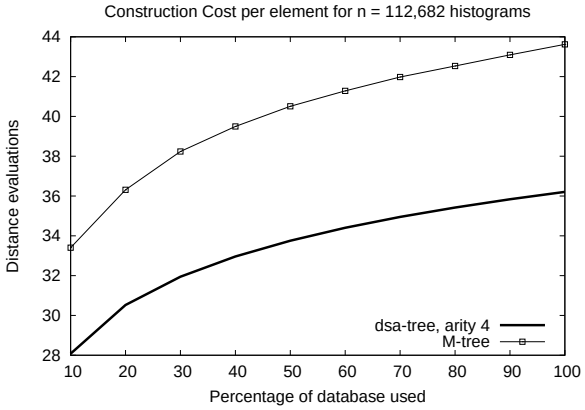
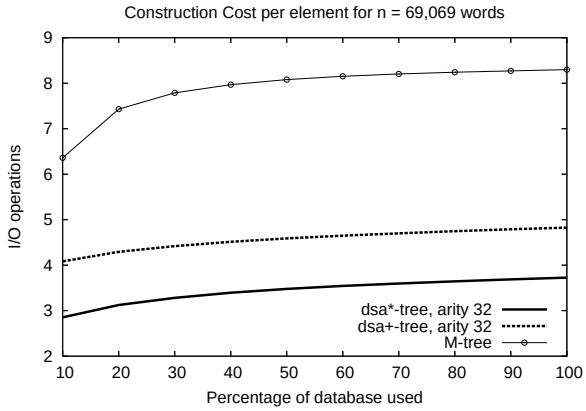
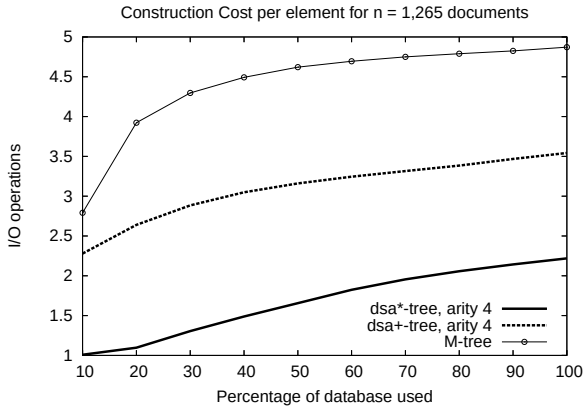
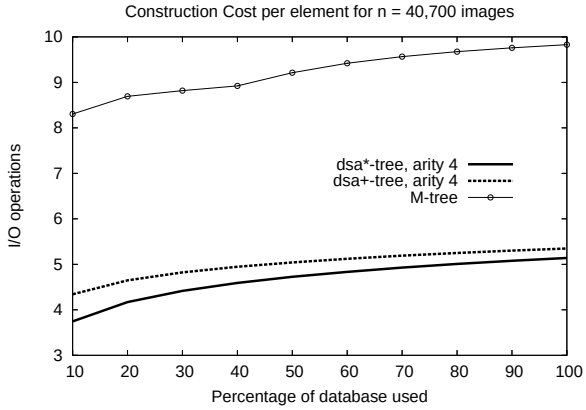


Figure 5. Distance evaluations at construction, for the four spaces.







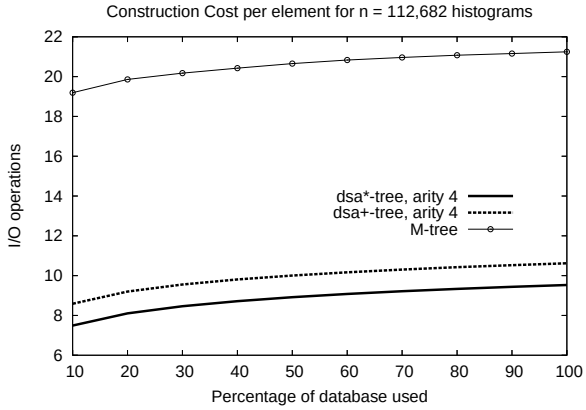
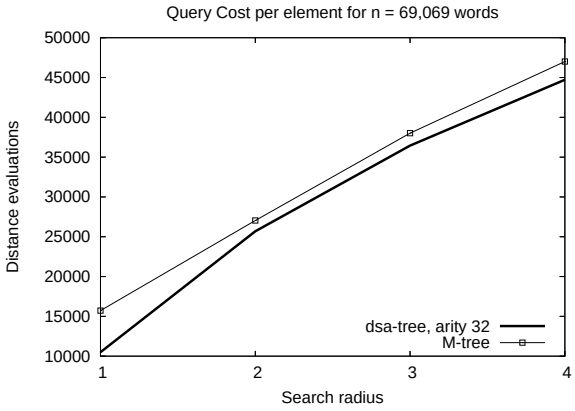
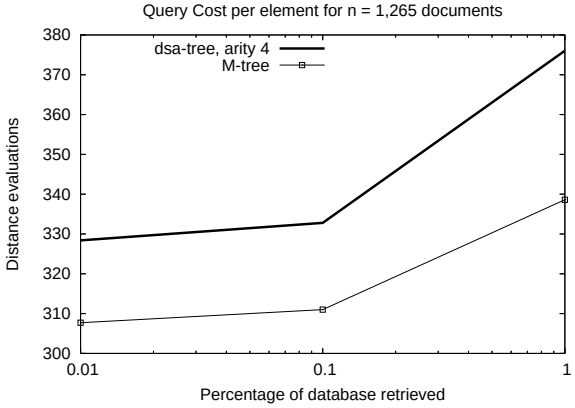
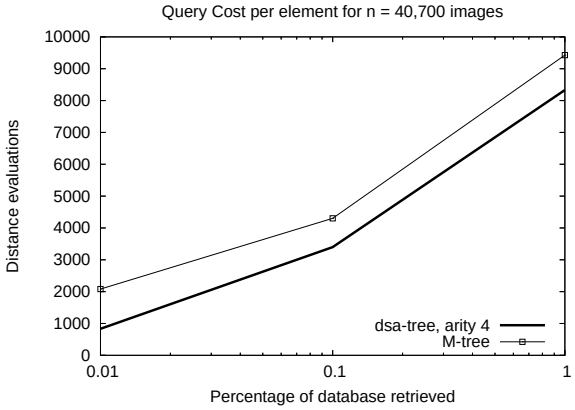


Figure 6. Number of disk page I/Os at construction, for the four spaces.







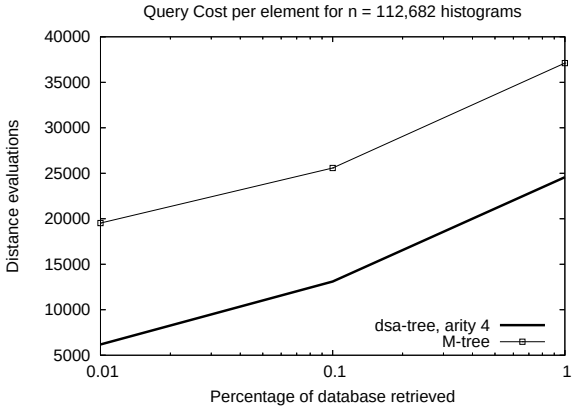
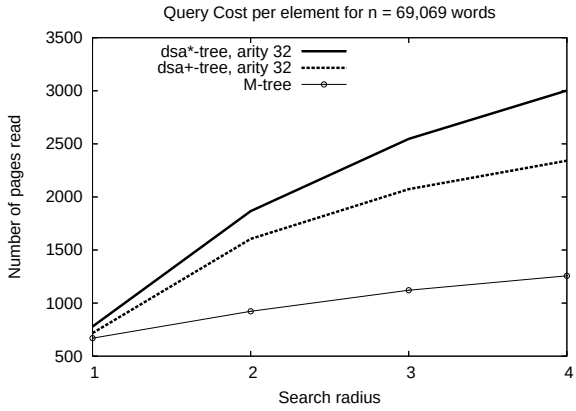
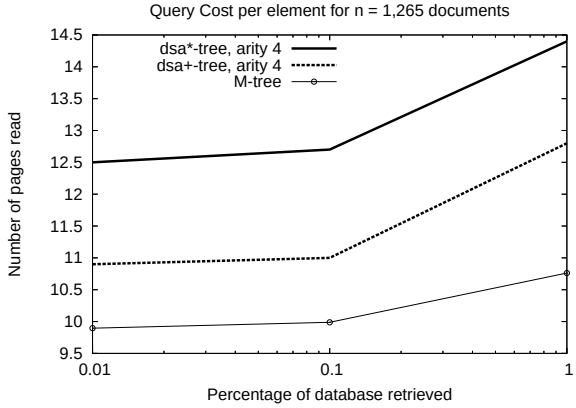
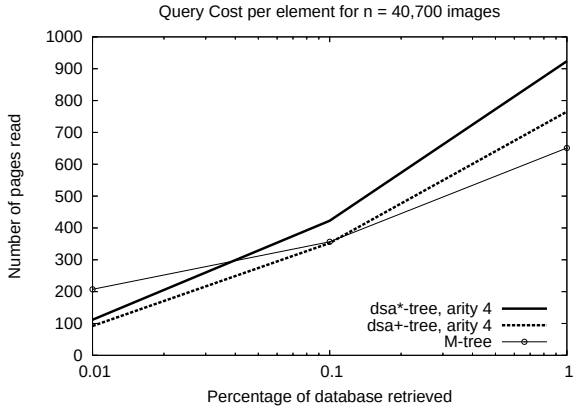


Figure 7. Distance evaluations at search time, for the four spaces.







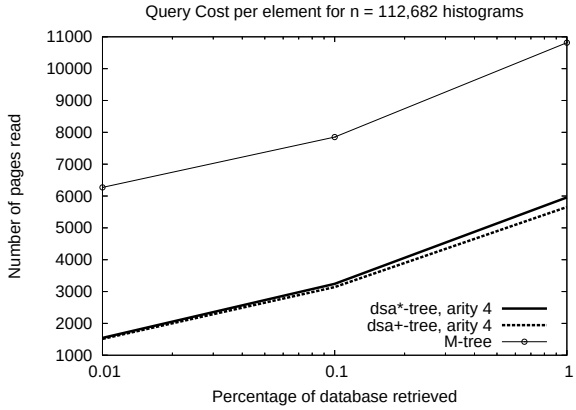


Figure 8. Number of disk pages read at search time, for the four spaces.