University of Alberta

Benchmarking Access Structures for High-Dimensional Multimedia Data

by

Nathan G. Colossi and Mario A. Nascimento

Technical Report TR 99-05
December 1999

DEPARTMENT OF COMPUTING SCIENCE
University of Alberta
Edmonton, Alberta, Canada

Benchmarking Access Structures for High-Dimensional Multimedia Data

Nathan G. Colossi
Institute of Computing, State Univ. of Campinas, Brazil
colossi@dcc.unicamp.br

Mario A. Nascimento
Department of Computing Science, Univ. of Alberta, Canada
mn@cs.ualberta.ca

Abstract

In multimedia databases it is usual to map objects into feature vectors in high-dimensional spaces. In order to speed query processing access structures, or indices, are required. Unfortunately, in the case of similarity queries, which are in fact nearest neighbor queries, classical spatial access structures such as the R*-tree are bound to fail when the space dimensional is not low. Fortunately, on the other hand, several access structures for high dimensional spaces, e.g., the SS-tree, SR-tree and M-tree have been proposed. However, each of those structures have been benchmarked in a rather ad-hoc manner. This paper benchmarks and compares all above structures using a real dataset of 40,000 high-dimensional objects. All structures have been implemented on top of the GiST infrastructure to minimize the risk of implementation bias. Even though no structure can be claimed to be the undisputed winner, we have found that the SR-tree presents the best overall results.

1 Introduction

Actual multimedia databases are becoming more and more common. In fact, it is not unusual to have the WWW itself being considered a extremely large, though unstructured database. As is the case with "traditional" databases, query processing in multimedia databases can be improved considerably using indices. Multimedia data objects in general may be (and usually are) mapped into feature vectors in high-dimensional spaces. In the case of images, for instance, one may use color histograms as an image abstraction. In this case, each color is regarded as one spatial dimension. Hence, for each of the D colors used the number, or ratio if normalized, of pixels of color C_i is used as the i-th coordinate of the D-dimensional feature space. Despite some well known arguments against it, color histograms are widely used to represent and index images. Due to the lack of space we must refer the reader to [16] for a in-depth discussion on this issue.

As we shall discuss shortly, "classical" indexing structures for D-dimensional data ($D > 2$), e.g., the R*-trees, are not well suited to cope with medium to high values of D. In fact, it is not unusual to have images abstracted as 64 (or higher) color histograms. Fortunately, several access structures for high dimensional spaces, have been proposed, e.g., the SS-tree [17], SR-tree [12] and the M-tree [5]. However, they have not been all compared against each other under the same circunstances, in fact, the M-tree has not been compared to any structure other than the R*-tree. Thus, a general conclusion about the "best" structure can hardly be drawn. The goal of this paper is therefore to present benchmarking results we have obtained using all the above four structures, and a real dataset of 40,000 images abstracted by means of their color histograms. We aim at provinding a means for multimedia database implementors to better appreciate, and thus make a better and more informed choice when implementing, indexing structures.

Towards that goal, this paper is structured as follows. Section 2 presents a brief overview of the investigated access structures. Next, in Section 3, we discuss the experimental setup used to obtain the results presented and discussed in Section 4. Section 5 concludes the paper, highlighting our main contributions.

It is important to note that, even though we constrain ourselves to using and discussing image datasets, we do so for the sake of exemplifying the use of high-dimensional multimedia data. The arguments presented should hold for other types of data as well.

2 Indexing High Dimensional Data

In 1984 Guttman proposed the R-tree [7], which has been the most referenced indexing method for spatial data. The R-tree is a balanced data structure designed for secondary memory, which abstracts data objects into minimum bounding rectangles (MBRs—note that a point in space is a degenerate MBR). The MBRs are grouped in an hierarchical manner, within overlapping MBRs. A spatial query, such as an returning all MBRs which intersect with another reference MBR, is processed by travessing the R-tree top-down (refer to [7] for details). The R-tree is also self-organized, providing support for dynamic insertion and deletion of data items (MBRs). The R-tree's main purpose is to provide an efficient filter to eliminate most but not necessarily all of the unwanted response. Due to the MBR abstraction, the actual response is obtained after refining the dataset returned by the R-tree. This has proved to be a rather acceptable overhead when compared to scanning over all data items regardless of where the query is spatially posed.

Several authors improved on the R-tree's main strategy, and the R*-tree [1] is one of most efficient R-trees to date. Its chief component is the concept of deferred node splitting. A node split occurs when the insertion algorithm ends up chosing a tree node which is filled to its capacity. The idea is to avoid a node split by removing and re-inserting some of the entries in the node about to be split. This has proved to be a clever and cost-effective way of improving the R-tree's performance.

Unfortunately the R-tree family is not well suited to index high-dimensional spatial data. The reason is that each entry in the tree nodes must keep the coordinates for the MBR associated to that particular entry. For instance, for a D-dimensional MBR with sides aligned to the coordinated axis, each entry must record coordinates of D opposite corners of the rectangle. Assuming a system of real coordinates, that would imply $2 \times D$ real numbers. To that one must also add the space for a pointer to the descendent MBR. As one can easily foresee, the larger the indexed dimension, the more space each entry will require. This goes on to the point where so few entries fit in a tree node that the resulting tree is rather deep instead of shallow, as ideally desired. This is of the aspects of the "dimensionality curse" and causes the index to become rather inefficient. For an interesting discussion on the "curse" refer to [3].

This "curse" has motivated a great deal of research, specially as the demand to store and query multimedia, i.e., potentially high-dimensional, data became a reality in the last few years. Among the several structures proposed we will briefly review in the following (and benchmark later): the SS-tree [17], the SR-tree [12] and the M-tree [5]. Other structures proposed, though not investigated in this research are: the TV-tree [13], the X-tree [2] and, more recently, the Slim-tree [11], the LSD^h-tree [9], and the Hybrid-tree [4].

The TV-tree uses an interesting strategy to "zoom" in the most important dimensions of the dataset, allowing one to take advantage of the data semantics. The X-tree resorts to using "super-nodes" (which are basically variable sized nodes) to avoid splitting nodes in higher dimensions. The Slim-tree, like the M-tree, indexes metric spaces, and uses a clever node splitting strategy to diminish node overlapping, which is generally the main cause to degrade query performance. Finally, The Hybrid-tree mixes ideas from both data-partitioning index structures, such as the R-tree family of structures, and from space-partitioning index structures, e.g., the K-D-B-tree [14], whereas the LSD^h-tree builds on the "low-dimensional" original LSD-tree [10] and shows that in some cases the "fan-out" of a tree node may be independent of the indexed dimensionality, which is a quite desirable feature.

The SS-tree is similar to the R*-tree, with a striking difference however. Instead of MBRs, it uses minimum bounding spheres (MBSs), which are centered in the centroid of the points contained in a given subtree, to represent objects. Therefore, instead of $2 \times D$ real numbers to represent a D-dimensional MBR, only $D + 1$ such numbers are needed, D for the sphere center and one for its radius. This savings in space grows very fast with the indexed dimension. The most important consequence of this is not the savings in space, but rather the fact that more entries can fit in a tree (disk) node, making the SS-tree tree a shalower structure, hence, speeding up query processing. The SS-tree is otherwise so similar to the R*-tree that it also uses the concept of deferred split and the query algorithms originally designed for the R*-tree also work with minimal changes on the SS-tree. It has been shown that the SS-trees outperforms the R*-tree using large synthetically generated datasets and a small real dataset of 100-dimensional objects.

The SR-tree's inventors noted that using minimum bounding spheres does have some advantages, however. Spheres can have a much larger volume than "equivalent" rectangles, leading to increased ratio of MBR overlap, which ultimately reduces query performance. To overcome this the SR-tree uses both rectangles and spheres, hence leading to less overlap between the indexed regions (now a combination of MBRs and MBSs). Another advantage of this strategy is that the SR-tree copes better with the non-uniformness of the data set. Even though the SR-tree is more costly to build, it was shown to outperform the SS-tree (and consequently the R*-tree).

The M-tree uses a quite different approach than the previous structures. Instead of the indexing spatial locations, such as MBRs or MBSs and their hierarchies, the M-tree indexes adimensional metric spaces. Given a search space (possibly amorphous) all the M-tree requires is a formal definition of a distance function between objects which observes the triangle inequality. The M-tree algorithms then are able to construct a balanced tree which may be more or less efficient depending on the accuracy of the distance function. Note that knowledge about the data semantics may play an central role in the M-tree's query performance. Besides the possibility of several distance functions, several node split policies have been devised and that has been one of the main investigated issues in the M-tree development so far. In this paper we will use the M-tree split policy which appears to be the best one so far, namely the minMax with parent confirmation and L_2 metric for the histogram distance. (refer to [5] for details). Although it has not been thoroughly tested using high-dimensional data it seems to be a good candidate structure, mainly because it does not suffer from the "dimensionality curse" as defined above. However, unlike most indexing structures, the M-tree may be CPU-bound instead of I/O-bound. The distance function can easily become complex, and slow down search and update performance with the increase in the space dimensionality. Furthermore, it has not been compared against other indexing structures besides the R*-tree.

3 Experimental Setup

One common pitfall when comparing results based on different implementations is to determine how much of the good (or bad) results are due to good (or bad) engineering at the source-code level. This is specially difficult to assert when the source code is provided by third parties. It is trivial to give examples where clever (poor) implementation may lead to extremely good (bad) results, which is likely to unintentionally bias comparisons. In an attempt to avoid that issue we used the Generalized Search Tree (GiST) framework [8]. As better stated in GiST's own WWW site: "*The GiST is an extensible data structure, which allows users to develop indices over any kind of data, supporting any lookup over that data. This package unifies a number of popular search trees in one data structure ... To make a GiST work, you just have to figure out what to represent in the keys, and then write 4 methods for the key class that help the tree do insertion, deletion, and search.*[1]. The four methods are needed for: maintaining the resulting tree consistent; consolidating tree nodes; measuring the effect of node update; and assigning the distribution of data items once a node split must occur. As of the time of this writing, the current GiST version, 2.0, already included the source code for the R*-tree, SS-tree and SR-tree. The authors

[1] `http://gist.cs.berkeley.edu/gist1.html`

of the original M-tree implemented it using an earlier version of GiST (version 0.9). We obtained and modified it to use the structure in GiST's Version 2.0. Even though one cannot guarantee that implementation biases will not be present, we believe this issue is minimized once all investigated structures use the same underlying structure.

It is rather common to see access structures being benchmarked in a *ad hoc* manner using synthetically generated datasets. The problem in this approach is generating meaningful data sets, e.g., images with realistic color distributions, is not a trivial task. As a result, some structures have been compared using high-dimensional data uniformly distributed in a hypothetical feature space. This is hardly the case in real life scenarios. In this paper we use a set of 40,000 color images, i.e., real color pictures from a commercially available stock CDROM. We believe that this provides a feature vector set which resembles more closely the actual distribution of color in a general scenario. Indeed, as we shall see, using a uniformly distributed dataset may lead to very different results, which we believe cannot be used for comparisons, as the dataset itself is likely to be flawed.

The image set is processed and their colors are mapped into the HSV color model [16] (though this is irrelevant for our purposes) and quantized in such a way that we obtain three distinct datasets, using, 16, 32 and 64 colors, respectively, from the same initial dataset. This will allow us to compare each structure with respect to the dimensionality of the dataset.

The tree node in most access structures is directly linked to the disk page size and most published research has used node sizes of 4 Kb. Not too long ago, Gray and Graefe [6] presented arguments indicating that current index pages should probably be 16 Kb large. As a matter of fact in the near future, pages of 8 Kb may be considered too small given the predicted throughput of future I/O systems. The effect of page size is seldomly investigated in the indexing literature, and therefore we evaluated the investigated structures using page sizes of 4, 8 and 16 Kb.

When the number of dimensions was varied in the experiments that follow, the page size was kept constant at 8Kb. Conversely, when the page size varied, the dimension of the data set was set at 32. For all tests, the query we used to benchmark the structures was a 21 nearest-neighbors query [15]. In the case of the dataset we used, that would be equivalent to providing one sample image from the dataset and searching the index for the 20 images most similar to that one. It is important to stress though that we are not concerned with the quality of the answer. This depends heavily on way the images are processed, and this is not the focus of this research. Instead, we are only concerned with the quantity of resources consumed by the structures when indexing and querying high-dimensional data. For that, we need not inspect the answer set but rather the resources consumed to obtain it.

Finally, the hardware used in our experiments was configured as follows: a dedicated stand-alone Pentium II Class CPU, running Linux at 300 MHz, and using 192 Mb of RAM as well as a large hard disk on a SCSI interface. All query processing times reported are averages obtained over 150 nearest-neighbor queries, over 10 differents trees, where each tree was build using a random order of the data set.

4 Results Obtained

Unlike most performance studies we have not used the number of disk I/O as our metric, but rather the actual processing time. This is due to the fact that some structures, notably the M-tree may be CPU-bound instead of I/O bound. Furthermore, given the low-load environment we had, we were able to verify that no memory swap was needed and thus the reported processing time for I/O bound processes should be proportional to the actual I/O time.

Due to the lack of space we will not show the results obtained using uniformly distributed datasets, nonetheless, it is worthwhile noting that in most cases the R*-tree becames one of the best, if not the best structure. We consider this a serious mislead, and given that real data is hardly uniformly distributed, the results obtained here should be more appropriate to be used as a general indicator of performance.

Figure 1 shows clearly how the R*-tree suffers as the dimension increases. As argued earlier, the higher the dimensions the smaller the number of entries per tree node and therefore the more nodes are needed. As reported

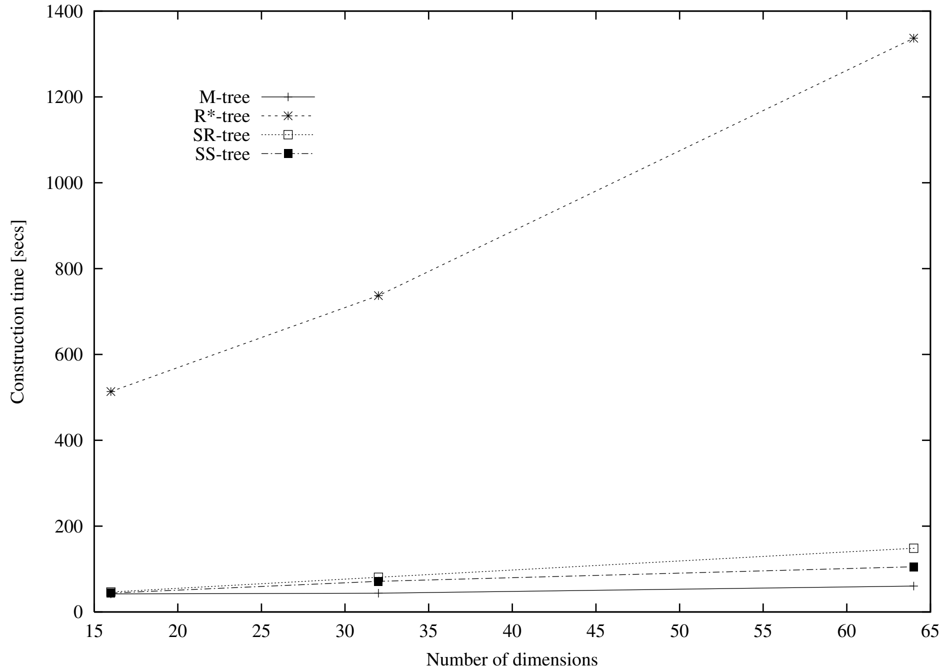

Figure 1: Index construction time versus number of dimensions

in [12] the SR-tree really required more time than the SS-tree (about 50%). The M-tree was the faster structure, being about 40% faster than the SS-tree. It is importante to note that the use of a more complex distance metric could change this considerably.

When we varied page sizes (Figure 2) all structures but the R*-tree (which was the worse structure by far) had nearly the same qualitative behavior, with the M-tree being the fastest (requiring between 36 and 56 secs) and the SR-tree being the slowest (requiring between 79 and 84 secs). The larger variance in the M-tree comes from the fact that larger nodes require more distance computations. This is an information that would probably not be transparent should we have reported only the number of I/Os.

Figure 3 shows the nicest feature of the SR-tree, which is the least sensitive access structure in terms of query processing time when the indexed dimension increases. It seems to indicate that the additional time spent in constructing the SR-tree pays off at query time, which is arguably a good trafe-off. For small dimensions even the R*-tree yields acceptable performance. After about 32 dimensions the M-tree becomes slower than the SS-tree, this is due to the CPU-bound distance calculation part of the index traversal, but again, a more informed distance metric could change the slope of the M-tree's curve.

It is again worthwhile noting that when using uniform distributed data the R*-tree behavior is so much different that it is overall the best structure !

Increasing the page size (Figure 4) is specially benefitial to the R*-tree, as larger nodes yield a shallower tree. It is interesting to note that larger page sizes are more benefitial to the M-tree than to the SS-tree. Even though one might think that larger page sizes increase the M-tree's computational effort (which is indeed true) this increase is not as severe as in the case above where the number os dimensions increase. Also the M-tree structure results in better clustering when more node space is available, thus the observed decrease in query time.

Figures 5 and 6 confirm the fact that the SS-tree is indeed very compact, and reveal that the M-tree is not as space-efficient as the other ones. Particularly it cannot take advantage of larger nodes, whereas all other can, especially the SR-tree. If disk pages continue to grow as predicted in [6] the SR-tree is also bound to become very space efficient.

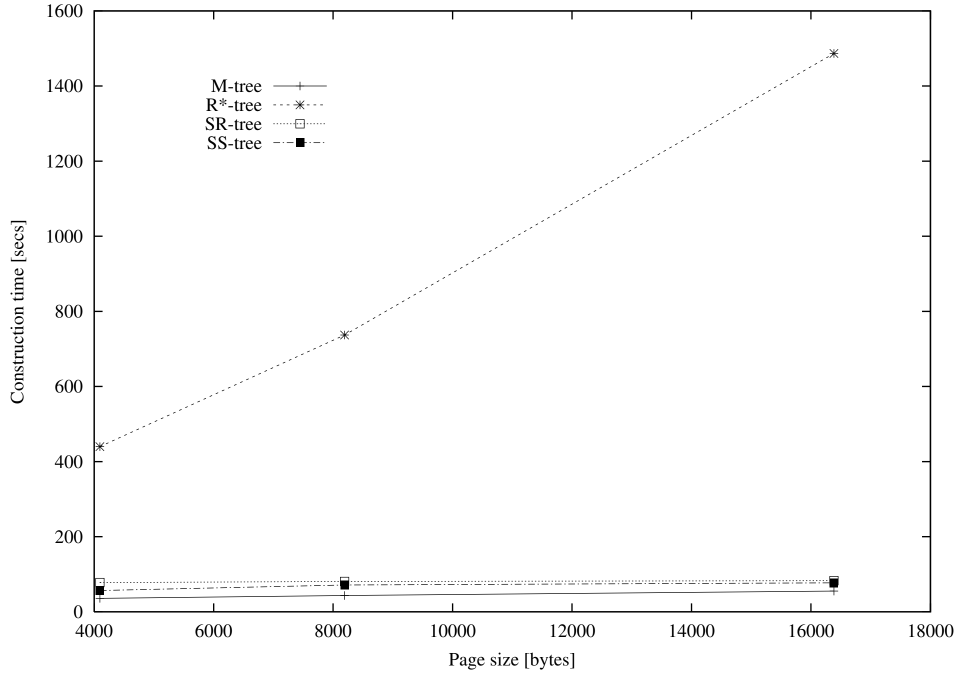

Figure 2: Index construction time versus disk page size

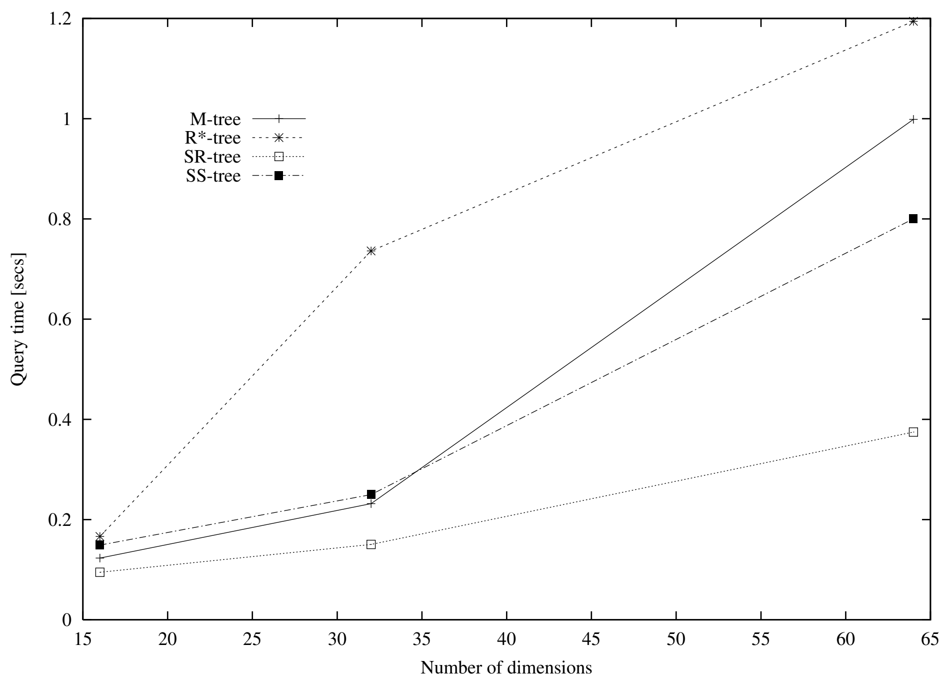

Figure 3: Query processing time versus number of dimensions

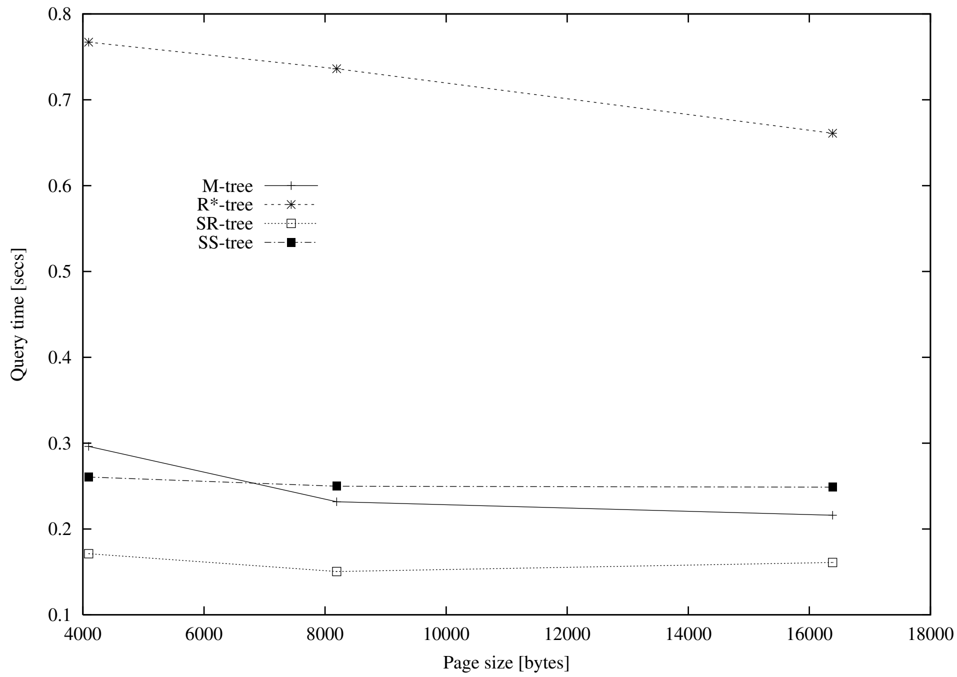

Figure 4: Query processing time versus disk page size

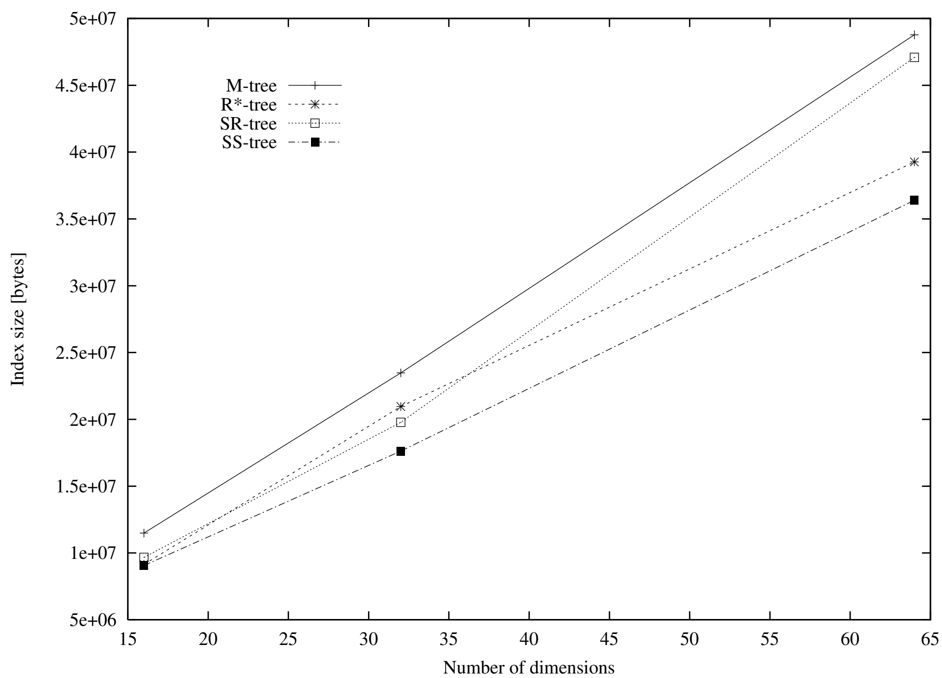

Figure 5: Index size versus number of dimensions

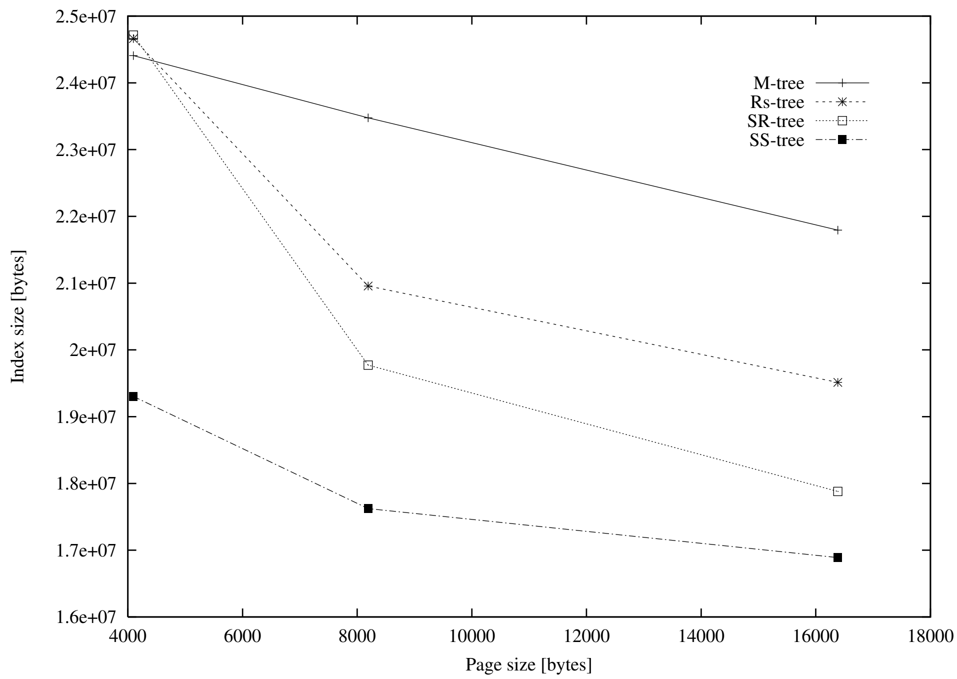

Figure 6: Index size versus disk page size

5 Conclusions

We presented the motivation for indexing high-dimensional (multimedia) data and the related problems that arise when "traditional" spatial access structures are used. We also reviewed, albeit briefly, some recent proposals to tackle this problem, namely, the SS-tree, SR-tree and M-tree.

The main contributions of this paper are as follows. To the best of our knowledge, this is the first direct comparison of all these access structures using the same real data set of non-trivial dimensionality. Indeed, this also seems to be the first time the M-tree was compared to structures other than the R*-tree. It has also investigated the effect of page sizes, which has been a somewhat neglected aspect, despite the astonishing evolution of database I/O systems. Overall, the structure of choice, i.e., more robust and resilient, seems to be the SR-tree. One must note however, that the M-tree is indeed promising as it does not rely only on Euclidean metrics, but rather it leaves definition of the distance functions open for the user.

We plan to extend this benchmark study to investigate the effect of: the initial spatial distribution of the objects; the size of answer set; few different node split and re-organization policies for the M-tree; and finally, include the newer structures (e.g., Slim-tree and SR-tree) in the benchmark.

Acknowledgment

Nathan G. Colossi was supported by a graduate fellowship from CNPq, Brazil. Mario A. Nascimento initiated this research while at the State Univ. of Campinas and is currently partially supported by a Startup Research Grant from the Univ. of Alberta. The authors thank Marco Patella for the comments and for providing the source code for the M-tree implemented under GiST version 0.9. Norio Katayama's and Paul Iglinski's suggestions for improvements on an earlier version of this paper were also appreciated.

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. of the 1990 ACM SIGMOD Intl. Conf. on Management of Data*, pages 322–331, 1990.

[2] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. of the 22nd Intl. Conf. on Very Large Data Bases*, pages 28–39, 1996.

[3] K. S. Beyer et al. When is "nearest neighbor" meaningful ? In *Proc. of the 7th Intl. Conf. on Database Theory*, pages 217–235, 1999.

[4] K. Chakrabarti and S. Mehrotra. The hybrid tree: An index structure for high dimensional feature spaces. In *Proc. of the 15th Intl. Conf. on Data Engineering*, pages 440–447, 1999.

[5] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. of the 23rd Intl. Conf. on Very Large Data Bases*, pages 426–435, 1997.

[6] J. Gray and G. Graefe. The five-minute rule ten years later, and other computer storage rules of thumb. *ACM SIGMOD Record*, 26(4):63–68, 1997.

[7] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the 1984 ACM SIGMOD Intl. Conf. on Management of Data*, pages 47–54, 1984.

[8] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for databases systems. In *Proc. of the 21st Intl. Conf. on Very Large Data Bases*, 1995.

[9] A. Henrich. The LSD^h-tree: An access structure for feature vectors. In *Proc. of the 14th Intl. Conf. on Data Engineering*, pages 362–369, 1998.

[10] A. Henrich, H.-W. Six, and P. Wildermayer. The LSD-tree: Spatial access to multidimensional point and non-point objects. In *Proc. of the 15th Intl. Conf. on Data Engineering*, pages 45–53, 1989.

[11] C. Traina Jr. et al. Slim-trees: High performance metric trees miniminzing overlap between nodes. Technical Report CMU-CS-99-170, School of Computer Science, Carnegie Mellon University, 1999. To appear at the Proc. of the 7th Intl. Conf. on Extending Database Technology.

[12] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. of the 1997 ACM SIGMOD Intl. Conf. on Management of Data*, pages 369–380, 1997.

[13] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *The Intl. Journal on Very Large Data Bases*, 3(4):517–542, 1994.

[14] J.T. Robinson. The K-D-B-tree: a search structure for multidimensional dynamic indexes. In *Proc. of the 1981 ACM SIGMOD Intl. Conf. on Management of Data*, pages 10–18, 1999.

[15] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. of the 1995 ACM SIGMOD Intl. Conf. on Management of Data*, pages 71–79, 1995.

[16] J. R. Smith. *Integrated Spatial and Feature Image Systems: Retrieval Analysis and Compression*. PhD thesis, Graduate School of Arts and Sciences, Columbia University, 1997.

[17] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. of the 12th Intl. Conf. on Data Engineering*, pages 516–523, 1996.