Similarity Indexing with the SS-tree *

David A. White
dwhite@cs.ucsd.edu

Ramesh Jain
jain@ece.ucsd.edu

Visual Computing Laboratory
University of California, San Diego
9500 Gilman Drive, Mail Code 0407
La Jolla, CA 92093-0407

Abstract

Efficient indexing of high dimensional feature vectors is important to allow visual information systems and a number other applications to scale up to large databases. In this paper, we define this problem as "similarity indexing" and describe the fundamental types of "similarity queries" that we believe should be supported.

We also propose a new dynamic structure for similarity indexing called the similarity search tree or SS-tree. In nearly every test we performed on high dimensional data, we found that this structure performed better than the R^*-tree. Our tests also show that the SS-tree is much better suited for approximate queries than the R^*-tree.

1 Introduction

In many fields, there is a need for what we call similarity indexing. The goal of similarity indexing is to facilitate efficient similarity queries of a dataset of typically high dimensional feature vectors. Similarity queries are queries that are related to some measure of similarity between feature vectors. One common example is a query for the most similar feature vectors given a reference feature vector.

Our primary application area for similarity indexing is visual information systems [1, 2]. Visual information systems encompass image, multimedia, interactive video, medical, and 3D databases. For example, a similarity query on a content-based image database could find pictures with a similar color, texture, or shape [3, 4, 5]. Other applications include time series indexing (ie. financial databases) [6], DNA databases [7], CAD databases [8], case-based reasoning in AI [9], and information retrieval [10, 11].

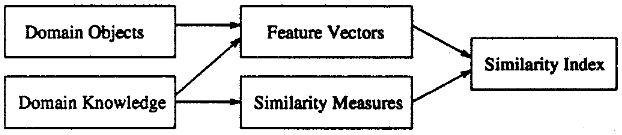


Figure 1: Illustration of dependencies in the creation of a similarity index

2 Similarity Indexing

The primary goal of similarity indexing is the same as other indexing methods: to minimize average and worst case time required for query operations. In addition, structures that support dynamic updates (insertions and deletions) and that have efficient disk-based implementations are preferred since many applications require these features to scale to large databases.

There are three components of similarity indexing that differentiate it from other types of database indexing. The data or objects being indexed are represented by medium or high dimensional feature vectors, usually between 5 and 100 dimensional or higher. Most spatial indexing structures are not designed to handle data of this dimensionality. Second, the feature vectors are queried primarily in terms of one or more measures of similarity or dissimilarity. Typically, dissimilarity between feature vectors will be measured in terms of some metric. The metric properties (positivity, symmetry, and triangle inequality) are usually basic assumptions made when designing and optimizing similarity indexing structures. The most common metric is Euclidean distance or weighted Euclidean distance, although other metrics such as city block distance can also be used. Lastly, the types of queries that are performed are fundamentally different than queries performed on standard indices.

It is important to mention that finding the appropriate representation for domain objects as feature vectors, and defining the correct similarity measures

*This research was funded in part by the National Science Foundation under Grant #MIP 9420099

1063-6382/96 $5.00 © 1996 IEEE

on those feature vectors is an area of research in itself. Usually the knowledge of a domain expert in the application area is required (figure 1). The role of similarity indexing, then, is to provide a low level mechanism that is useful in many application domains.

2.1 Similarity Queries

The types of query operations that will be performed on a similarity indexing structure fall into three categories: similarity selection operations, similarity join operations, and similarity sample operations. In this section, we choose to ignore the issue of the actual definition of similarity in hope that our definition will apply to similarity indexing structures using different similarity measures. We also speak of "maximum similarity" instead of "minimum dissimilarity" although implementations usually deal with the latter. Also, queries are not limited to one similarity measure as long as the similarity indexing structure allows multiple similarity measures. In fact, the similarity measure might be another parameter used when specifying a query.

The *Similarity Selection* operation will simply "Find objects similar to a reference." This query is analogous to a selection operation performed on relational databases. However, as it is, the query statement does not specify how many similar objects should be retrieved, but it seems reasonable that a similarity selection could retrieve either the k most similar objects (k-nearest neighbors) or all objects with a threshold level T of similarity. In fact, the two type of queries can be generalized into one type of query.

Stated precisely, a similarity selection operation always returns the k most similar objects to the reference object. Usually the objects returned will be sorted in order of decreasing similarity. In cases where multiple objects have the same similarity to the reference object (to machine precision), the ordering of returned objects and exactly which objects will be returned is undefined. Two thresholds are specified on the query. The first threshold k specifies the maximum number of objects returned by the query, and a threshold T specifies the minimum threshold similarity required in order for an object to be returned by a query.

If the threshold T is set to the smallest possible level of similarity, then the query becomes a standard k most similar query. If k is set to its maximum value, then the query returns all objects with a threshold level of similarity.

In many applications, especially interactive applications, because of the nature of high dimensional data, it will often be possible to trade a small probability of an inaccurate result for a large performance improvement. We and other researchers working on the nearest neighbor problem have observed this [12]. Therefore, in many applications, it will be appropriate to have one additional parameter that specifies the maximum error allowed in the final result. Usually, this parameter will be a bound ϵ in the range $0.0 \leq \epsilon \leq 0.5$ that specifies that an algorithm can ignore searching regions in feature space as long as those regions can not contain objects within distance $(1-\epsilon)D_{\max}$, where $D_{\max}$ represents the "spherical" region in feature space that needs to be searched to guarantee an exact result. Of course, the query is exact if $\epsilon = 0$.

The *Similarity Join* operation will "Find all pairs of object that are similar." This type of query could be considered a generalization of a join operation. A typical relational-style join operation retrieves (and combines) rows or objects with exactly matching attributes, while this operation retrieves rows or objects with "similar" attributes. The same issues mentioned above for the similarity selection apply here also, and there are additional issues.

Finally, the *Similarity Sample* operation will "Find a representative sample of objects (similar to a reference)." This query operation has no analog in relational operations, but nonetheless may be the most important and most frequently used operation because of its applicability in browsing the contents of a database.

A similarity sample operation will retrieve samples of the objects in the database relative to a reference object, or relative to the complete contents of the database. In general, a query must specify the reference object, some measure of the spread of the sampling, usually based on the measure of similarity, and the number of samples desired.

There are a number of different ways to define a sampling of a region of the database. Samples could be representative of the population of the database, so the spacing between samples in a region should be approximately inversely proportional to the density of samples in that region. Samples could be approximately equidistant from each other within the regions spanned by the database. Samples could be centers of "clusters" of objects in the database, using some clustering criterion. In this case, the clustering might be performed be separately in parallel with indexing or perhaps the clustering within the indexing structure could be used. Finally, samples could be even chosen at random from all objects within the region being sampled. The best sampling method(s) may be a tradeoff between efficiency and accuracy.

Similarity selection could be considered a special case of the sample operation, since a sampling of the database with a small enough spread should intuitively return the most similar objects to a reference object. However, most implementations will likely consider these types of queries different operations.

Because queries might combine similarity related information with standard constraints based on alphanumeric information associated with an object, a general purpose implementation of a similarity indexing query interface should also support the specification of a predicate function (or functions in the case of the similarity join). The predicate function determines whether the query is satisfied by a similarity query result. For example, suppose we have a content-based face database [13, 14], and we have an unidentified picture of someone who is in the database, we may want to determine who that person is. We could search the database using only pictorial information, but if we know other information about the person such as his/her first name, sex, or approximate age, we can increase the accuracy of the search by also specifying this information. If predicates are not supported, we can never know for sure how many people should be requested by a similarity selection or similarity sample operation, because we do not know *a priori* how many of the returned people will satisfy our constraints.

Finally, in some applications, it will be useful to find the most dissimilar objects. It is easy to reformulate the above operations in terms of dissimilarity and call them *dissimilarity selection*, *dissimilarity join*, and *dissimilarity sample* operations.

2.2 Applications and Examples of Similarity Queries

This section should help explain some of the applications for similarity indexing and clarify how the queries described above will be used in applications. First we will explain some ideas about how users might use a database that employs similarity indexing, since the paradigm is much different than that of the standard relational databases. The SQL query mechanism has been very successful because in most cases the following assumptions hold true:

1. The users know what they are looking for when they issue a query.

2. The users can specify a query (in SQL or via a user interface) that will retrieve what they want.

However, in many applications that employ similarity indexing, these assumptions will usually not hold true. This is because the feature vector representation is used primarily in applications where standard methods are ineffective or can be improved upon. Usually this means that the objects being indexed can not be described easily or concisely using a textual representation. Therefore, even if users are aware of what they want, it may be difficult to precisely describe what they are looking for. Further, the users will often only have a foggy idea of what they want, or may just be interested in exploring the contents of the database. Researchers working in application areas for similarity indexing are well aware of this problem [3, 10].

Therefore, we make a distinction between two modes a user can be in when accessing a database interactively. For completeness, we add a third mode which is typically non-interactive.

In *locator mode*, users do know what they are looking for, but as was mentioned above, may have difficulty expressing their query to the system, or they may not be aware of how similarity is understood by the system.

In *browsing mode*, users either are only interested in exploring the database or have only a foggy idea what they are looking for.

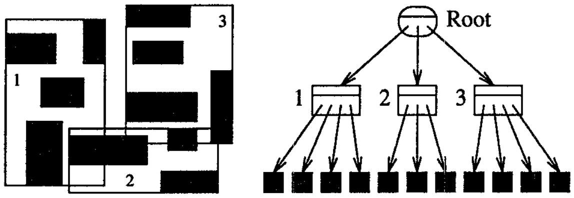
Finally, in *analysis mode*, the goal is to analyze the contents of the database in some way. The similarity indexing structure will typically be queried multiple times under control of a program or SQL query. These queries are typically non-interactive, although speed is still an important issue.

Users in all three modes might use either the similarity selection or similarity sample operation, but the similarity join operation will usually be only an analysis mode operation. However, when browsing large databases, the primary query type will probably be the similarity sample operation, because, with the appropriate interface, it can allow the user to "zoom in" and "zoom out" of the database with respect to a reference object, where zooming in and out would decrease or increase the spread of the sampling. In addition, the measure of similarity could be modified interactively in order to change the type of sampling.

2.3 Similarity Indexing Requirements and Tradeoffs

In this section, we explain some assumptions about requirements and tradeoffs we used when designing our similarity indexing structure. The assumptions are based on our experiences with visual information systems, but we believe these assumptions will hold in many or most applications that require similarity indexing:

Query performance is much important than update performance. In image and multimedia databases,



(a) Representation in 2D feature space

(b) Tree representation used in memory or on disk

Figure 2: The R-tree structure

the database will typically rarely be updated, and often the updating will be done as an offline processing step. In contrast, query performance must be as fast as possible since similarity queries will typically be performed online, interactively, and will account for most accesses to the indexing structure.

Disk access counts alone do not adequately measure performance. Since queries in visual information systems are interactive, databases of relatively small size (thousands of objects) are typically memory resident to allow fast response time. In the future, even large databases may be memory resident. Therefore, the performance of index structures should be measured by total CPU time (and perhaps other measures) in addition to the number of disk accesses (leaf and internal/directory nodes).

Dynamic updating of the database should be supported, but query performance should still approach that of a static structure. There is no question that a dynamic structure are preferred over a static structure because of its greater flexibility. However, static structures or structures that are reorganized or optimized typically allow more efficient query processing than purely dynamic structures. We suggest that future work should look at effective "packing" algorithms [15] and algorithms which optimize an existing dynamic index [16].

3 Related Work

Most previous work in the database literature has focused on indexing lower dimensional data and on other types of queries besides similarity queries. The *k-d* tree was one of the first structures proposed for indexing multidimensional data for nearest neighbor queries [17]. Recently, this structure has been used in geographic information systems for queries like similarity queries [18], and might be useful for similarity indexing. Other methods such as space filling curves, linear quadtrees [19], and gridfiles [20], do not scale well to high dimensions, but may be useful for medium dimensional data.

The R-tree [21] (figure 2) and its most successful variant, the R*-tree [22], have been used most often for indexing high dimensional data in the database literature. However, since ranges are stored on each dimension, the index requires more space and time to search in higher dimensionality. For this reason, higher dimensional data typically is mapped to a lower dimensional space before indexing in R-trees [4, 23].

The TV-tree [24] is the only method in the database literature thus far that has been proposed specifically for indexing high-dimensional data. Performance comparisons clearly show that the TV-tree can be much more efficient than the R*-tree. However, the improved performance depends on two assumptions. The first assumption is that dimensions of the feature vectors are ordered by "importance." The second assumption is that sets of feature vectors in the dataset will tend to exactly match on dimensions, especially on the first few "important" dimensions.

The first assumption is reasonable (if not desirable) since an appropriate transform may be used. The second assumption was not explicitly stated in the paper, but a careful analysis of their algorithms reveals that their performance improvement depends upon it. In some applications, the original feature vectors contain a small set of discrete quantities, so the second assumption does hold.

Unfortunately, this second assumption will normally not be true in visual information systems, and in many other applications. Features in these applications are typically real-valued, so that chances of exactly matching on dimensions is negligible. In this case, the TV-tree reduces to an index on only first few dimensions. Small changes in the proposed algorithms should allow the TV-tree to be a modest improvement over the R*-tree in these applications. However, in this paper, we will refer to the R-tree (and variants) as the best previously known structure for similarity indexing because it has proven itself in more similarity indexing applications.

There is also related work outside of the database literature. In the information retrieval literature [10, 11], work has been done on cluster files [25] that proposes structures similar to the SS-tree. In the image database community, a static indexing structure based on Kohonen nets was suggested [26]. There is also related work in the computational geometry and vector quantization literature [12].

4 A Similarity Indexing Structure: The SS-tree

In this section, we first explain the definition of similarity used by the SS-tree and then describe the

SS-tree structure and a few implementation issues.

4.1 Similarity

When defining similarity indexing, we did not precisely define similarity since we were not dealing directly with implementation issues, but in this section we define similarity in the context of the SS-tree. However, we believe this definition of similarity will most commonly be used. Here we usually speak of dissimilarity or distance between feature vectors rather than similarity. The distance measure is a weighted Euclidean distance metric. (Of course, *squared* weighted Euclidean distance is used when possible to avoid costly square roots.) This similarity measure was chosen primarily for efficiency reason, as will be explained below. It is defined mathematically as follows:

$$D(\mathbf{x},\mathbf{y}) = \sqrt{(\mathbf{x}-\mathbf{y})^T \mathrm{diag}(\mathbf{w})(\mathbf{x}-\mathbf{y})} \qquad (1)$$

where $\mathbf{x}$ and $\mathbf{y}$ are the feature vectors being compared, and $\mathbf{w}$ is a vector representing the relative weight of each dimension for distance. Every element of $\mathbf{w}$ must be nonnegative.

This distance metric was chosen because it has a property that the distance metric can still be calculated after any invertible linear transform of the feature vectors. This property is true of Euclidean-based metrics (based on the L_2 norm), but does not hold for other metrics based on the city block distance (L_1 norm) or maximum value metric (L_∞ norm).

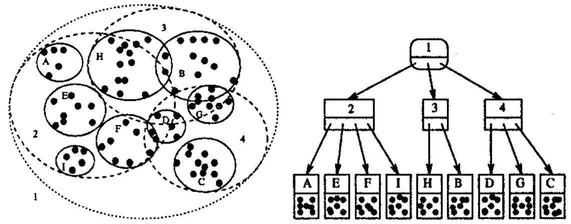
In order to use the SS-tree, we rely on a domain expert to help in the indexing process, as was mentioned before (figure 1).

1. Feature vectors must be provided in a format such that a weighted Euclidean distance metric on those features can approximate the desired measure of dissimilarity.

2. Knowledge of the domain should be used to constrain the types of similarity measures between feature vectors that will normally be used in queries. This knowledge can then be used to tune the performance of the SS-tree (or multiple SS-trees).

Each SS-tree structure uses one base distance metric (dissimilarity) defined by a vector $\mathbf{w}_b$ of size N_b (the feature vector size). In order to save space and avoid exhaustive searches of the SS-tree, we require that every element of $\mathbf{w}_b$ be positive (zero weights are not permitted). In other words, we require that all elements in the feature vector be used for indexing.

At the time of SS-tree creation, we also require that the domain expert specify groups of elements of the feature vectors that will always have a weight in the same proportion to each other. This is done by specifying an integer vector $\mathbf{g}$ of size N_b containing group labels of each vector where group labels can vary from $1 \ldots N_q$, where N_q is the number of groups. Queries can then be specified conveniently in terms of a query weight vector $\mathbf{w}_q$ of size N_q. The actual weight of any feature vector element in a query is then the weight of its group times its base weight. The use of groups does not make the SS-tree less general purpose because each element of the feature vector could be in its own group (ie. $N_b = N_q$). However, when there are groups, the SS-tree can better optimize itself because it has more information about how weights can vary.



(a) Representation of high dimensional feature space in 2D

(b) Tree representation used in memory or on disk

Figure 3: The SS-tree structure

```
struct SSElem {
  SSElemPtr child_array_ptr; // Child pointer
  int immed_children;      // Children in array
  int total_children;      // Children in subtree
  int height;              // Height above leaf
  int update_count;        // w/out refresh values
  float radius;            // of enclosing sphere
  float variance;          // Sum squared dist.
  float centroid[DIM];     // FVect or Mean value
  char  data[DATA_SIZE];// Data repr. elem
};
```

Figure 4: The SSElem structure (// denote comments)

In some cases, when queries using different weights are common, one SS-tree may not be effective on its own, since a single SS-tree can only be optimized for one base distance metric. In this situation, provided that enough disk space and/or memory is available, multiple SS-trees with different base distance metrics could be used.

4.2 SS-tree Data Structure

The overall SS-tree data structure is shown in figure 3. Every disk node in the structure consists simply of an array of the SSElem structure in figure 4.

In our current implementation, both internal nodes and leaf nodes use the above structure. If the SSElem is a leaf element, data holds the data for that leaf, and centroid holds the object's feature vectors, and radius bounds the object's extent in feature space (this was zero in our tests on point data). If SSElem is an internal node, then its information is defined by its children. At all times, the values of child_array_ptr, immed_children, total_children, height and centroid contains the pointer to the child node (array), the size of that array, the total number of children, the node height above the leaves, and the centroid (mean value) of the child vectors. The radius is always greater than or equal to the distance to the furthest feature vector from the centroid, and is recalculated based on its children. The variance could be useful for approximate queries, but was not used in the tests presented here. The data holds the information for the closest immediate child to the centroid. This is obviously convenient for the similarity sample operation, because a sampling of the database can be achieved without accessing the the leaf nodes of the tree. The update_count is used to allow the values used by internal nodes to be periodically recalculated using its immediate children. This allows floating point errors to be avoided in higher level nodes due to loss of significance, and allows *lazy* recalculation of the radius and variance during the insertion process. Currently, each node's values are recalculated once every five times a node is changed.

4.3 Query Algorithm

Both our SS-tree and R*-tree search algorithms simply search regions in order of minimum distance from the query point until the query results are guaranteed correct (to the required accuracy). Our algorithms use two priority queues: a search queue and result queue. We do not provide details since search algorithms exist for the k-nearest neighbor framework for the R-tree [27] and for the *k-d* tree [18, 12]. We also do not describe an algorithm for the similarity sample operation, but a random sampling algorithm for the R-tree has already been suggested [28]. The same algorithm will work with the SS-tree. In addition, the SS-tree can provide a fast sampling using only internal nodes.

4.4 SS-tree Insertion Algorithm

Due to space consideration, we provide only a high-level description of the insertion algorithm. The insertion algorithm is similar to the R*-tree [22] in that it uses the concept of a *forced reinsert* and each node can have a minimum of m and maximum of M children (except the root node). Therefore, the first step of insertion is to add the inserted node to the reinsert list (linked list of SSElem), and enter a loop that inserts nodes from the reinsert list until it is empty. Hereafter, we will describe the insertion algorithm for a single node N from the reinsert list.

If only the root node exists (an empty structure contains only a root node), a new node array is created and N is inserted there (updating the root node appropriately). Otherwise, the first step is to descend the tree until the new parent of the node N is found. Every node traversed (including the parent) is updated by incrementing both the update_count (doing recalculation if needed) and total_children, and updating the centroid and radius appropriately (including adding the distance moved by the centroid to the radius). The subtree for insertion (within each array of SSElem) is simply the node whose centroid is closest to the centroid of N.

Once the parent is found, and it is checked to see if it is full (immed_children $= M$). If there is space, then immed_children is incremented and N is added to the parent's child array. If the parent is full, then if the parent's children have not already been reinserted, they are reinserted. Otherwise, the parent must be split. In order to be able to determine whether a node's children have been reinserted, a list of reinserted nodes' child_array_ptrs are maintained, since the child_array_ptr of a given node is unique and is never modified by an insert operation (unless the node is split).

The reinsert operation is similar to the R*-tree algorithm. The SS-tree uses the same reinsert fraction parameter (our tests used a value of .3, or 30% reinserted). Before the nodes are added to the reinsert list, they are deleted from all parents. This is done by simply calculating their combined centroid and total children, and deleting all from the parents with a single update to each parent's centroid and total_children. Of course, the distance moved by each parent's centroid must be added to the radius, and the direct parent's immed_children must also be updated.

The split algorithm simply finds the dimension with the highest variance, and then chooses the split location (only two choices when $M = 2m$) to minimize the sum of the variances on each side of the split. Then, two new parent nodes are created and initialized with the split elements. When the root is split, a new root array is allocated and the two parents are written to it, updating all the SSElem information appropriately. Otherwise, the node closest to its parent replaces the original parent. The other new parent node is reinserted (deleted from the tree and added to the reinsert

		SS-tree				R*-tree			
d	FO	Lvs	ULvs	LLf	Nds	Lvs	ULvs	LLf	Nds
2	83	3.670	2.253	2.416	3.87	3.235	2.804	2.927	5.35
3	62	8.105	3.689	4.364	5.94	10.60	6.166	8.005	13.8
4	101	14.10	4.338	5.626	8.31	15.33	7.263	9.908	10.7
5	84	29.04	5.961	9.505	14.8	42.90	12.66	26.00	19.6
6	71	60.39	7.612	15.03	26.0	83.51	17.52	50.73	31.6
7	62	110.0	9.408	22.68	35.6	217.2	28.77	140.7	59.1
8	55	193.8	11.19	33.38	54.3	301.2	33.58	200.1	68.3
9	50	324.3	13.03	46.66	65.8	507.0	39.67	351.5	114
10	45	523.7	14.90	66.84	73.0	659.1	42.10	497.3	123
11	41	843.7	16.53	92.64	102	794.5	44.90	608.6	152

Table 1: Uniform Distribution (size 100,000)

		SS-tree				R*-tree			
d	FO	Lvs	ULvs	LLf	Nds	Lvs	ULvs	LLf	Nds
2	83	4.299	2.492	2.776	9.90	5.212	3.071	4.139	6.93
3	62	13.19	4.784	7.105	22.8	29.83	10.22	19.5	27.8
4	101	40.57	6.596	17.43	16.0	55.09	11.34	32.13	15.2
5	84	107.9	9.508	40.51	24.0	212.7	21.95	132.7	41.1
6	71	229.0	12.10	73.14	32.0	462.7	31.19	288.1	60.4
7	62	428.5	14.94	119.0	41.0	694.0	37.39	502.5	83.8
8	55	723.2	17.58	164.7	54.0	1089	45.90	851.6	114
9	50	1139	20.03	212.2	65.0	1507	53.05	1191	145
10	45	1573	22.34	259.2	79.0	1924	57.99	1563	172
11	41	2072	25.79	336.1	98.0	2373	63.85	1957	215

Table 2: Gaussian Distribution (size 100,000)

		SS-tree				R*-tree			
d	FO	Lvs	ULvs	LLf	Nds	Lvs	ULvs	LLf	Nds
10	45	85.84	9.793	28.37	7.97	127.5	21.07	94.95	13.3
20	47	115.9	10.74	38.51	7.00	187.7	31.41	146.2	17.7
50	39	158.1	13.65	55.73	9.98	244.3	33.99	191.2	27.4
100	19	309.7	19.06	104.7	34.5	497.9	44.1	387.6	127

Table 3: EigenFace Dataset (size 7000)

list).

4.5 SS-tree vs. R*-tree Performance

Since query performance is our main goal, we are only presenting query performance results. However, in most cases, the SS-tree required significantly less CPU time (5-10x less) to insert the same number of elements because its choose subtree algorithm is linear rather than quadratic [22], although the SS-tree probably requires more disk accesses. The storage utilization of the SS-tree was 85% ± 1%, while the R*-tree utilization was 70-75%.

The results of our test are shown in tables 1, 2, and 3. The Uniform dataset in table 1 is in the range $[0, 1)$ on each dimension. The Normal dataset in table 2 has a mean of 0 and variance of 1 on each dimension. Both synthetic datasets consisted of 100,000 vectors. The EigenFace dataset in table 3 was computed from a database of 7000 faces [13]. The different values of d on the EigenFace dataset were obtained by truncating the original 100D vectors. We would normally recommend performing dimensionality reduction on real datasets before indexing, but doing so would be redundant on the EigenFace dataset.

All results are the average performance (over 1000 random trials) of a 21-nearest neighbor query relative to a point in the dataset, which we hope will approximate the expected performance of a 20-nearest neighbor query relative to a new point in the distribution. The results were all for exact queries $\epsilon = 0$, although we also show information that should help predict approximate performance. The "*d*" column shows the dimension of the dataset. The "FO" column is the fanout of leaf nodes. For the SS-tree, this is also the fanout of internal nodes, and for the R*-tree it is about double the fanout of internal nodes. The "Lvs" column shows the number of leaf nodes accessed during an exact query. The next two columns provide information that should indicate the usefulness of approximate queries. The "ULvs" column shows the average number of used leaf nodes (leaves accessed that changed the state of the results). The "LLf" column shows the average last useful leaf accessed. All additional leaf accesses (on average LLf - Lvs) do not change the results but are required to guarantee the correctness of the query. Intuitively, the "LLf" count should be a good indicator of approximate query performance. The "Nds" column shows the number of internal nodes accessed. CPU time results were not shown here due to time and space constraints.

The results show that on higher dimensional data (> 5D), the SS-tree provides faster query performance than the R*-tree in almost every test. The results also show the performance degradation with dimension of both structures. One possible way to overcome this problem is to use approximate queries instead of exact queries. Although we do not provide performance results on approximate queries, the tests indicate that for approximate queries, the SS-tree should provide much better performance and more accurate results.

5 Conclusion

In this paper, we propose a definition for a new type of indexing we call "similarity indexing," and provide a solution in the form of a new indexing structure called the SS-tree. Included in the definition are important concepts such as the similarity sample operation. We suggested a dynamic similarity indexing structure, the SS-tree, and compared it with the R*-tree. The results of our tests suggest that the SS-tree is superior for similarity indexing applications. In future work, we plan to look at using *a priori* knowledge of the dataset to further improve performance of similarity indexing structures, because, as was previously mentioned, using such knowledge is practical in most similarity indexing applications.

Acknowledgments

We thank Shankar Chatterjee for his many helpful comments and suggestions while revising this paper, and Amarnath Gupta for his helpful comments and for suggesting the idea of the *locator* and *browsing* modes. We also thank Bradley Horowitz and Sandy Pentland for providing the EigenFace dataset, and Stefan Berchtold for providing R*-tree code.

References

[1] R. Jain, "InfoScopes: Multimedia Information Systems," in *Multimedia Systems and Techniques* (B. Furht, ed.), ch. 7, pp. 217–254, Kluwer Academic Publishers, Norwell, MA, 1996. To be published. Also available as Visual Computing Lab Technical Report VCL-95-107.

[2] A. Gupta, T. Weymouth, and R. Jain, "Semantic queries with pictures: the VIMSYS model," in *Proc. 17th International Conference on Very Large Data Bases*, pp. 69–79, Sept. 1991.

[3] V. N. Gudivada and V. V. Raghavan, "Content-based image retrieval systems," *IEEE Computer*, vol. 28, pp. 18–22, Sept. 1995.

[4] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, *et al.*, "Efficient and effective querying by image content," *Journal of Intelligent Information Systems: Integrating Artificial Intelligence and Database Technologies*, vol. 3, pp. 231–62, July 1994.

[5] A. Pentland, R. Picard, and S. Sclaroff, "Photobook: Tools for Content-Based Manipulation of Image Databases," in *Proceedings of the SPIE: Storage and Retrieval for Image and Video Databases II, San Jose, CA*, vol. 2185, pp. 34–47, Feb. 1994.

[6] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast Subsequence Matching in Time-Series Databases," in *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pp. 419–429, June 1994.

[7] S. F. Altschul, W. Gish, W. Miller, E. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, pp. 403–410, Oct. 1990.

[8] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "Fast searching for partial similarity in polygon databases." Submitted for publication at SIGMOD '96.

[9] C. K. Riesbeck and R. C. Shrank, *Inside case-based reasoning*. Hillsdale, NJ: Lawrence Eribaum Associates, 1989.

[10] G. Salton and M. McGill, *Introduction to Modern Information Retrieval*. McGraw Hill International Company, New York, 1989.

[11] C. Faloutsos, "A Survey of Information Retrieval and Filtering Methods," Tech. Rep. CS-TR-3514, Dept. of Computer Science, Univ. of Maryland, Aug. 1995.

[12] S. Arya and D. M. Mount, "Algorithms for fast vector quantization," in *Proc. of DCC '93: Data Compression Conference* (J. A. Storer and M. Cohn, eds.), pp. 381–390, IEEE Press, 1993.

[13] M. Turk and A. Pentland, "Eigenfaces for recognition," *Journal of Cognitive Neuroscience*, vol. 3, no. 1, 1990.

[14] J. R. Bach, S. Paul, and R. C. Jain, "A visual information management system for the interactive retrieval of faces," *IEEE Transactions on Knowledge and Data Engineering*, vol. 5, pp. 619–628, Aug. 1993.

[15] I. Kamel and C. Faloutsos, "On packing R-trees," in *Proc. 2nd International Conference on Information and Knowledge Management (CIKM-93)*, (Arlington, VA), pp. 490–499, Nov. 1993.

[16] D. M. Gavrila, "R-tree index optimization," in *Advances in GIS Research* (T. Waugh and R. Healey, eds.), Taylor and Francis, 1994. Also, CS-TR-3292, University of Maryland, College Park, 1994.

[17] J. H. Friedman, J. H. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software*, vol. 3, pp. 209–226, Sept. 1977.

[18] A. Henrich, "A distance-scan algorithm for spatial access structures," in *2nd ACM workshop on Advances in Geographic Information Systems*, (Gaithersburg, Maryland), pp. 136–143, Dec. 1994.

[19] H. Samet, *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[20] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The grid file: an adaptable, symmetric multikey file structure," *Proceedings of the ACM TODS*, vol. 9, pp. 38–71, Mar. 1984.

[21] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pp. 47–57, June 1984.

[22] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," in *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pp. 322–331, May 1990.

[23] J. Hafner, H. Sawhney, W. Equitz, M. Flickner, *et al.*, "Efficient color histogram indexing for quadratic form distance functions," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, pp. 729–736, July 1995.

[24] K.-I. Lin, H. Jagadish, and C. Faloutsos, "The TV-tree - an index structure for high-dimensional data," *VLDB Journal*, vol. 3, pp. 517–542, Oct. 1994.

[25] G. Salton and A. Wong, "Generation and search of clustered files," *ACM Transactions on Database Systems*, vol. 3, pp. 331–346, Dec. 1978.

[26] H. Zhang and D. Zhong, "A scheme for visual feature based image indexing," in *Proceedings of the SPIE: Storage and Retrieval for Image and Video Databases III, San Jose, CA*, vol. 2420, pp. 36–46, Feb. 1995.

[27] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," in *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, (San Jose, CA), pp. 71–79, June 1995.

[28] M. Ester, H.-P. Kriegel, and X. Xu, "A Database Interface for Clustering in Large Spatial Databases," in *Proceedings of the 1st International Conference on Knowledge Discovery and Data Mining (KDD-95)*, Aug. 1995.