*Supported by the
National Science Foundation
Engineering Research Center
Program (NSFD CD 8803012),
Industry and the University*

Diamond-Tree: An Index Structure for High-Dimensionality Approximate Searching

by C. Faloutsos and H.V. Jagadish

TR 92-97

Diamond-Tree: An Index Structure for High-Dimensionality Approximate Searching

Christos Faloutsos† *H. V. Jagadish*

University of Maryland *AT&T Bell Laboratories*
College Park, MD *Murray Hill, NJ*

ABSTRACT

A selection query applied to a database often has the selection predicate imperfectly specified. We present a technique, called the *Diamond-tree*, for indexing fields to perform similarity-based retrieval, given some applicable measures of approximation. Typically, the number of features (or dimensions of similarity) is large, so that the search space has a high-dimensionality, and most traditional methods perform poorly. As a test case, we show how the Diamond-tree technique can be used to perform retrievals based on incorrectly or approximately specified values for string fields. Experimental results show that our method can respond to approximate match queries by examining a small portion (1%-5%) of the database.

1. INTRODUCTION

Requirements for approximate matching arise in several applications. For instance,

- Searching for names or addresses, say in a customer (mailing) list, where these are partially specified or have errors. For example ''1234 Springs Road'' instead of ''1235 Spring Rd'', or ''Mr. John Smith'' instead of ''Dr. J. Smith, Jr.''
- Searching bibliographic entries with possible errors or abbreviations in the various fields. For example the bibliographic query ''aho ulman tods 1979'' should retrieve the journal article [*Optimal Partial Match Retrieval When Fields are Independently Specified* A.V. Aho and J.D. Ullman, ACM Trans. on Database Systems, 4, 2, pp. 168-179, June 1979] despite the misspelling of the string "Ullman" and the use of the abbreviation "tods". Similar approximate matching facilities would be useful for searching manual pages etc.
- Traditional text databases, *e.g.*, law or patent office cases [15], where searches are performed for similar previous cases.
- Spelling error correction [3, 8, 27], as well as typing and OCR error correction [22]. There, given a wrong string, we should search a dictionary to find the closest strings to it.
- DNA databases [2] where there is a large collection of strings from a four-letter alphabet (A,G,C,T); a new string has to be matched against the old strings, to find the best candidates.

† Also with the Systems Research Center (SRC). This research was partially sponsored by UMIACS, by the National Science Foundation under the grants IRI-8719458 and IRI-8958546, by a Department of Commerce Joint Statistical Agreement JSA-91-9, by a donation by EMPRESS Software Inc. and by a donation by Thinking Machines Inc..

- Query by image content, where both shapes in a database and the query shape are described in terms of some feature vectors that provide an approximate indication of shape [19]. Alternatively, the features could be the coefficients of the Discrete Cosine Transform, which is used in the JPEG image compression standard [35].
- Fingerprints are beginning to be digitized, and there is a need to retrieve fingerprints in a database that (approximately) match a specified query fingerprint, usually one lifted from the scene of a crime. While good techniques exist for matching fingerprints [26], there is no good indexing technique currently available.

In all of the above examples, there is a requirement of tolerating errors in specification and/or supplying a few nearest matches. The ability to perform approximate match retrievals is crucial to our ability to provide a user-friendly interface to such queries. In fact, it can even serve as the basis of a ''co-operative answers'' system.

In this paper, we propose a technique for indexing fields in databases, so that approximate searches are facilitated. The basic idea is to map attribute values into vectors in a ''feature space'', in which ''similar'' objects are close to one another. Then, we construct a multi-dimensional index structure on this feature space. The main example will consider string fields, although the proposed file structure can be applied to any type of field for which such feature vectors can be defined.

The paper is organized as follows: Section 2 describes the requirements of the target applications. Section 3 presents the proposed method and its manipulation algorithms, Section 4 gives the experimental results of our method on a real database. Section 5 provides a brief survey of approximate matching methods, which mainly apply to low dimensionality spaces. Section 6 lists the conclusions.

2. ENVIRONMENT - GOALS

A fundamental requirement for our indexing methods is the existence of some good feature extraction functions. These functions should be closely related to the (dissimilarity) distance function between two attribute values. For example:

- if the attributes are words, and the errors are OCR or typing errors: 26 features, each representing the count of the corresponding letter (all letters folded to lower case). E.g., "cab" becomes the vector (1, 1, 1, 0, ..., 0). Alternatively, we could use successive, overlapping triplets as features; in this case the feature vector would have 26^3 entries (if we optimistically consider only lower-case alphabetic characters - no digits, apostrophes etc.)
- if the field is a bitmap, it can be mapped to its first n coefficients of the Discrete Fourier Transform, according to the JPEG proposed standard [35]
- In Information Retrieval [30] a document can be mapped to a vector of V elements, where V is the vocabulary of the collection of documents. A "0" entry indicates absence of the corresponding term in the document, a non-zero entry (usually, "1"), indicates the importance of the term in this document.

In general, we rely on an expert to derive some good feature extraction functions for a given applications. Ideally, the distance in feature space should be related as closely as possible to the ''difference'' of the two objects. In order to derive the specifications for the target indexing method, we propose to classify the above applications according to the following characteristics:

- dimensionality of feature vectors: low (<4) / high
- sparseness of feature vectors
- Query types: *Fully specified* approximate matching vs *partially specified.* The former means that all the feature values are specified; the latter means that the query "does not care" about some of the feature values. An example of the latter is "find all the customers with last name *Mac Donald*", where *John Mc Donald* and *Mr. Michael McDonald Jr* should be retrieved.

We focus on applications with high dimensionality, sparse feature vectors, with fully specified similarity queries: For example, consider the setting of the dictionary look-up for error correction, with triplets as features. There, we have $\approx 10^4$ features, with $\approx$10 non-zero features per entry. Similarly, there is a consensus [20] that index structures for similarity retrieval in high-dimensional spaces is a key research issue in image databases.

Often, not all dimensions in feature space are independent. In this case it may be possible to reduce the dimensionality of the space by determining correlations between the different dimensions and finding an orthogonal basis. In fact, it has been suggested [7] that one may actually improve the performance of similarity metrics if the number of dimensions is reduced in some carefully controlled manner, since spurious information (noise) is more likely to be eliminated than real correlation. We intend to study the use of such dimension reduction strategies in the future. The point to note for the present is that even after dimension reduction, the dimensionality of the reduced feature space is still expected to be high, necessitating the use of an index structure such as the proposed Diamond-tree.

For the problem we are examining, each of the known methods presents problems: inversion on each feature could be inefficient, requiring the merging of long lists of postings to derive a short answer set; "fault tolerant" full scanning [21] could be expensive for large databases; common multi-dimensional access methods like the R-tree [13] and the grid-file [25] suffer from the high-dimensionality, as we discuss in Section 5. We can thus develop a set of requirements for the index structure we seek:

- It should be able to index high as well as low dimensionality spaces.
- It should efficiently support a variety of queries including exact match, approximate match and range queries.
- It should be able to work with disk-based files.
- It should work in a dynamic environment, supporting insertions and deletions efficiently, without the expensive construction or reorganization procedures.
- It should provide a good space utilization.

3. THE DIAMOND TREE INDEXING TECHNIQUE

Given that objects are mapped to vectors in a n-dimensional space, we have to define a distance measure for these vectors. In general, the Minkowski metric L_p between two vectors $\vec{v}$ and $\vec{u}$ is given by

$$L_p(\vec{v},\vec{u}) = \left[\sum_{i=1} | v_i - u_i |^p \right]^{1/p} \qquad (1)$$

The L_2 metric is the familiar Euclidean distance, which defines a sphere of approximation around a point. A simpler metric is the L_1 metric, which is the *Manhattan distance* measure that we use. In this case, the region of approximation around a point has a ''diamond'' shape; this is where the name of proposed data structure comes from. For example, in the 2-dimensional feature space of Figure 1(a), everything inside the diamond **A** is r units away from the center P of the diamond. The reasons we have chosen the L_1 distance are (a) a diamond covers less volume than a sphere of the same radius; thus, there will be fewer overlaps (b) it is intuitive, reducing to the Hamming distance for binary values (c) it is simple and fast to compute.

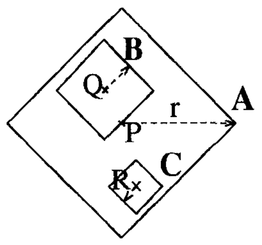


Figure 1(a): Illustration of a Diamond-tree in 2 dimensions.

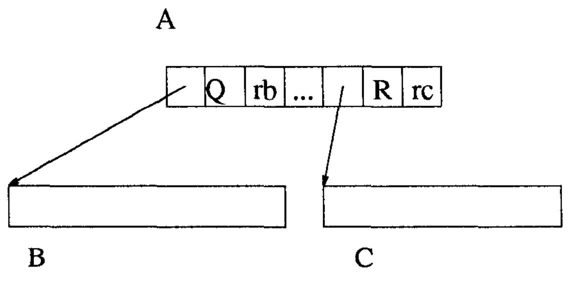


Figure 1(b): the Diamond-tree of Figure 1(a), on disk.

The basic structure is similar to a B-tree, in that each node ''covers'' a partition of space (a diamond) and has children that each cover some portion of this partition. Every node is stored in one disk page. Thus, the fanout is dictated by the page size. Diamonds are allowed to overlap.

Each diamond is described by its center and radius. The radius is a scalar. The center is a long, but sparse vector. We represent sparse vectors by identifying and recording only the non-zero values. Thus, each point is a set of pairs of the form (feature_id, value), or, in pseudo-C:

```
      struct POINT { list_of (int feature_id; float value) }
```

with one pair per non-zero coordinate. Similarly, a diamond is described by its center and the radius:

```
      struct DIAMOND { POINT center ; float radius}
```

A node in the Diamond-tree contains a set of diamonds, along with pointers to lower-level nodes:

```
      struct NODE { list_of ( DIAMOND d; NODEPTR ptr ) }
```

Each diamond is the minimum bounding diamond (MBD) for the corresponding child. At the leaf level, the pointers are null, and can be omitted. The definition of the MBD follows:

DEFINITION 1: The Minimum Bounding Diamond (MBD) for a set of diamonds is the diamond with the smallest radius that contains all the diamonds in the set.

In addition to its ability to handle high-dimensionality spaces, the Diamond-tree has several desirable features:

- It is balanced, with guaranteed space utilization of $\geq 50\%$.
- It can handle easily both searches and dynamic updates.
- It can be used for low dimensionality spaces, as well
- On-the-fly addition of new feature-identifiers is possible

Next we discuss each individual operation on the Diamond-tree, as a means to explaining its structure and its properties.

3.1. Search

Searching for a specific point, or all points within a specified region is similar to searching on an R-tree. Begin at the root of the tree and explore each child that has an MBD that intersects the specified query region (or includes the specified query point). Continue this process recursively until all such children nodes are explored. The MBDs of sibling nodes are not necessarily mutually exclusive. Even for a point query, it is possible that multiple indexing paths down to leaf nodes may have to be explored. The attempt is to use good clustering algorithms so that this overlap is minimized as much as possible. These aspects are similar to an R-tree.

Search for nearest neighbor is a little more complicated since the nearest neighbor could be arbitrarily far away and no single region query will do. We use a Branch-and-bound algorithm, similar to the one proposed in [12]. The idea is to traverse the tree, maintaining a cut-off distance and ignoring diamonds that are too far away. This cut-off distance is continuously updated, as we encounter diamonds that are more promising. Figure 2 illustrates the idea: Given the search point P and the diamond D1, there is no need to examine diamond D2.

```
/* prints the data points within 'tolerance' from 'p'
for the tree rooted at 'N' */

RSearch(POINT p, NODE N, float tolerance)
    for each diamond d in N
        if the diamond with center p and radius 'tolerance'
         intersects d then
            if N is a leaf, print d
            else RSearch( p, d, tolerance)
```

Range Search Algorithm

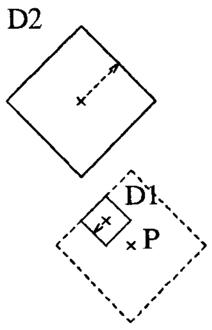


Figure 2: Illustration of the cut-off idea for nearest-neighbor search.

3.2. Insertion

To insert a point, first perform a search for the specified point location. The point is inserted in the leaf-node reached by the search. If more than one leaf node is reached by the search, then the point is inserted in any of them, using some rule to break the tie. In our implementation, we select the leaf node with center closer to the inserted point. An alternative is to choose the leaf node with least number of points.

If the point to be inserted is in some region of multi-dimensional space far away from existing points, there may be no leaf node with a minimum bounding diamond that encompasses the point. In this case, we must choose some leaf node in which to insert the point, and expand the MBD. In our implementation, this choice is the leaf node whose MBD radius is increased the least. An alternative is to choose the leaf node whose center is closest to the point being inserted.

Once the leaf node for the insertion has been determined, insert the point in this node, recomputing the MBD and the center of the node, if necessary. If the MBD of a leaf node is recomputed, this recomputation must be propagated upward until a node is reached whose MBD is not changed by the recomputation.

If a node overflows, upon the insertion of a point, the children of the node are divided into two groups. We distinguish two cases: (a) the insertion is a genuine insertion (b) the insertion is the result of a ''force-reinsert''.

In the former case, we split the overflowing node into two "natural" groups of diamonds. The two groups do not necessarily contain the same number of diamonds; the smaller group is ''force-reinserted''.

In the latter case, the node is divided into two groups of equal population. A minimum bounding diamond is found separately for each group, a new node created for each group, and these two new nodes inserted in place of the overflowed node.

In any case, the parent of the overflowed node is updated – if it overflows, this procedure is repeated recursively. If the root overflows, it is split in two as above, and a new root created as the parent of the two nodes resulting from the split, thereby increasing the depth of the tree by one.

```
Insert(POINT p, NODE root)
  Traverse the tree
  pick the correct leaf L
  insert p in L tentatively

  if L overflows AND it is a first-time insertion, then
    /* divide L into two "natural" groups */
    (L1, L2) = Split(L, 100%)
    if L1 has more diamonds than L2 then
      L = L1 ; force-reinsert the contents of L2
    else
      L1 and L2 are the results of the split of L

  if L overflows  AND it is a forced-reinsert then
    /* divide L into two equal-size groups */
    (L1, L2) = Split(L, 50%)
    L1 and L2 are the results of the split of L

  update the ancestor(s) of L
```

Algorithm Insertion

3.3. Splitting

As mentioned above, there are two cases in splitting, depending on whether the insertion that caused the split is a first-time insertion or a forced-reinsert. Both cases can be handled changing the x parameter, which specifies the maximum percentage of diamonds that could go to the largest group. Clearly, for x=50% we ask for two equally-populated groups; for x=100% there are no limitations - the largest group can be as large as the geometry of the specific node dictates.

Thus, the basic splitting routine is the same for both cases. Let C and R be the center and radius of the MBD of N, a node that has overflowed and has to be split. We would like to divide its children nodes into two groups, such that the total volume of the MBDs of the two groups is minimized (without having more than x% of the nodes in the largest group).

The idea is to find the two most remote children and use them as ''seeds'' for the groups G1 and G2. Then, each of the remaining diamonds is assigned to the group that will require the smallest radius increase.

To insert a diamond rather than a point (e.g., as a result of a split and forced-reinsert of a non-leaf node) the Algorithm Insertion can be used again, modified slightly to find a node at the appropriate level to insert into rather than a leaf node.

3.4. Deletion

Deletion of a point simply involves removing it from the leaf node that includes it. If this does not cause an underflow in the node, then the new minimum bounding diamond is computed, and after that there is no more work to be done. (We could propagate upward the recomputation of the MBD until the root is reached or the MBD does not change. However, if the reduction in MBD is small, this extra effort may not be worthwhile after each deletion, and could be batched, or delayed until some time when the system load is low).

If there is an underflow, then this leaf node is deleted and the points in it are ''forcibly'' re-inserted, using the standard insertion procedure described above. Deletion of a leaf node may cause its parent node to underflow, in which case the parent node is deleted and the diamonds in it reinserted, and so on recursively, until there is no underflow or the root node is reached.

3.5. Determining Minimum Bounding Diamond

In the case of the R-tree, calculating the minimum bounding rectangle is easy, and linear on the number of dimensions. In our setting, the problem is not as straightforward. Given a set of diamonds, the minimum bounding diamond can be found by solving the following linear programming problem:

PROBLEM MBD: Find the minimum bounding diamond for a set of diamonds.

Let there be n diamonds in the set, and d dimensions of the feature space.

Let the center of diamond i be the vector $\vec{x_i} = <x_{i1}, x_{i2}, \ldots, x_{id}>$

Let the radius of diamond i be r_i.

```
/* Splits node N into two new nodes, N1 and N2 */

Split( NODE N, float x) returns (NODE N1, NODE N2)

    Let 'd1' be the diamond farthest away from the center 'C'
       (Break ties arbitrarily).
    MBD1 = d1; N1 = { d1 };

    Let 'd2' be the diamond whose center is farthest away from the
    center of 'd1'.
    MBD2 = d2; N2 = { d2 };

    Loop, until all diamonds are assigned, or one of the new nodes
    has x% of the diamonds.
    •    Sort the un-assigned diamonds according to the increase in
         radius they will cause if assigned to N1, and if assigned
         to N2.
    •    Choose the diamond 'di' that results in the smallest ra-
         dius increase, and assign it to the appropriate node.
    •    Update the MBD of the node assigned to.

    Assign the leftover diamonds to the smaller node.
```

Algorithm Split

Let the MBD we desire have radius r and center $\vec{c}=<c_1, c_2, \ldots, c_d>$

We wish to minimize r, such that

For all $1 \le i \le n$,

$r \ge a_{i1} + a_{i2} + \cdots + a_{id} + r_i$

For all $1 \le j \le d$,

$a_{ij} \ge c_j - x_{ij}$

$a_{ij} \ge x_{ij} - c_j$

where the a_{ij} are intermediate variables representing the absolute value of the difference along dimension j of the points $\vec{c}$ and $\vec{x_i}$.

Thus we have a linear programming problem with $(n+1)*d + 1$ variables and $(2n+1)*d$ inequality constraints. This problem has size linear in n and d. Typically n is fairly small, being limited by the number of entries possible on a page, which is a few dozen at most. d could be large if all dimensions of feature space are considered. However, d can also be reasonable if we consider only the non-zero dimensions. Thus the problem size could be manageable.

Even though the linear programming solution described above is not prohibitive, it could still be too expensive to do every time an MBD has to be recomputed. A fast, approximate algorithm to calculate the MBD for a set of diamonds would be as follows:

ALGORITHM AMBD: Find an Approximate MBD, for a set of diamonds:

- Find the (weighted) average of the center coordinate vectors
- Perform any thresholding suggested (e.g., ignore features with values < 0.5 - see next subsection)
- Calculate the radius, so that it covers all the diamonds in the set.

The center of each diamond is found by calculating the weighted average of the vectors of the centers of constituent diamonds, weighting each diamond by the number of data points it includes. We used this approximate algorithm in our implementation.

3.6. Additional Implementation Considerations

During the implementation of the method, several "low-level" details had to be considered. We discuss them briefly, along with the proposed solutions

Rounding of the coordinates. Recall that no space is wasted recording zeroes of the largely sparse vectors that we expect to deal with. To minimize the number of non-zeroes that are to be recorded, it may be worthwhile to set a threshold below which any number will be treated as zero. This is particularly useful because the average (or sum) of several sparse vectors is no longer as sparse, with small non-zero components where only one (or a few) vectors were non-zero. If data points are boolean, one could additionally set a threshold above which any number will be treated as 1. If these thresholds are at 0 and 1 respectively, then the centers of diamonds are maintained exactly. If these thresholds are both set to 0.5, then we have a boolean vector for the center as well. It can be shown that such thresholding will decrease the average distance to the center of the points in a cluster. We have implemented the thresholding mechanism. The experiments mentioned in the next section were performed without thresholds, except that the number of non-zero features per point was (arbitrarily) limited to 12. Thus, only the 12 features with the largest weights were kept as diamond centers.

Distorted diamonds. Instead of using perfect diamonds, we could use "distorted" ones, by storing the extent along each non-zero attribute independently. That is, each diamond is stored as a set of triplets of the form

(feature_id, value, extent)

This is the Manhattan-distance equivalent of the Euclidean ellipses. However, the extra storage required will reduce the fan-out of each node; it is not clear whether the reduced volume of each diamond will offset the effects of the smaller fanout on the search performance. Thus, we didn't implement this version.

Dimension reduction. Finally, if the features are not independent, *dimensionality reduction* techniques could be used, to eliminate features that are highly correlated with others [7]. Given that these algorithms could be slow for a large data sets sampling could be used to derive quickly a (sub)-optimal set of independent features.

4. PERFORMANCE

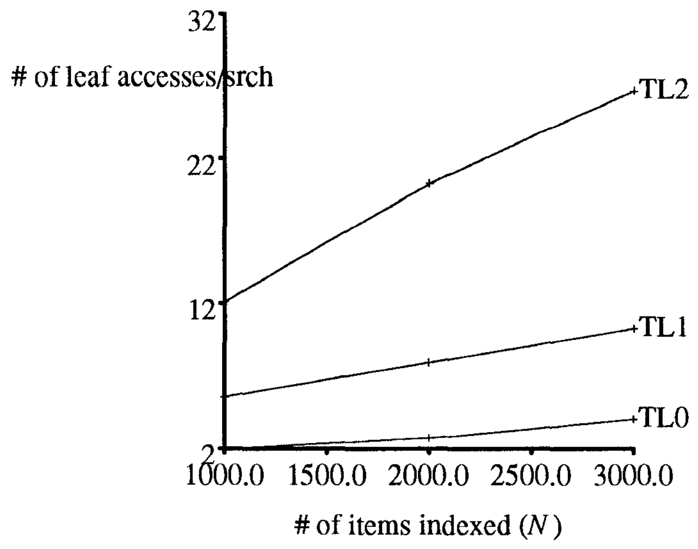
The code for the Diamond-tree was written in C and UNIX ^{TM}. The computational effort is measured in terms of the number of leaf pages fetched. The reason is that the root of an index tree, and sometimes even the next level of nodes, can be pinned in memory. Furthermore, in structures where searches may follow multiple paths down to the leaves, these paths usually share the same parents at some level below the root, and therefore it is useful to buffer in memory pages corresponding to the top few levels parents of the current node. As such, the number of leaf nodes accessed appeared to be the best measure of performance available.

To evaluate the performance of the Diamond-tree, we ran some experiments in performing approximate string matching using a Diamond-tree index. We used the first 1,000, 2,000 and 3,000 words of the UNIX dictionary (typically found in the file "/usr/dict/words"), after we deleted the strings with non-alphabetic characters, and folded the remaining strings into lower case letters. The page size was 1K, which roughly corresponds to 8 diamonds per page. The features were the letter counts, resulting in a 26-dimensions feature space. The queries were 254 words chosen at random from the above dictionary. We experimented with distance ("tolerance") ranging from 0 to 2. A distance of 0 represents an exact match query. A distance of 1 represents the possible addition or deletion of one character, and so on. Graph 1 shows the average number of leaves accessed per query, as a function of the database size N and the tolerance (TL). Graph 2 plots the same measurements, when the forced re-insert mechanism was turned off (dashed lines); for comparison, we plot with solid lines the corresponding performance *with* forced re-insert. Graph 3 shows the same results as Graph 1, plotting the leaf accesses as a function of the tolerance. The conclusions from the above graphs are the following:

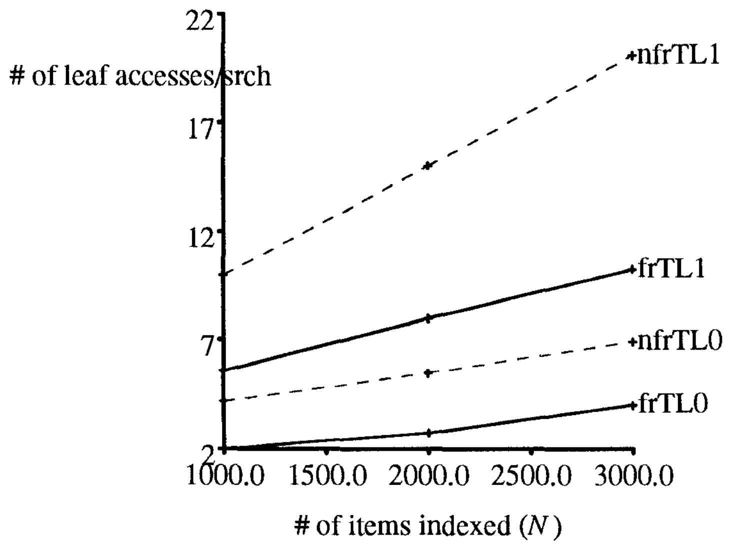
- Forced-reinsert pays off, possibly because it results in better trees
- Graph 3 shows that the response time of the method grows quickly (probably, exponentially) with the tolerance. This is expected, since the number of qualifying points increases quickly with the tolerance. In a n dimensional space, with points uniformly distributed, the number of neighbors within distance r is $O(r^n)$,
- The absolute number of disk accesses is very encouraging: even for the worst queries (tolerance=2), the Diamond-tree responds with $\approx$30 leaf accesses; assuming 20 msec for each access (the typical disk access time), we can expect response times $<$1sec.

Comparing the performance of the Diamond-tree with other methods is difficult: traditional methods (R-trees, grid-file etc.) are not applicable, due to the high dimensionality; the literature

UNIX ^{TM} is a registered trademark of UNIX System Laboratories.

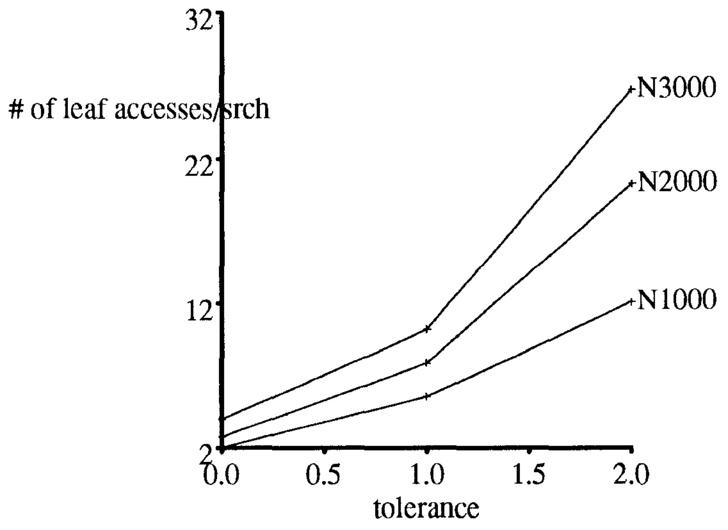


Graph 1: Avg. leaf accesses per query, as a function of number of words N. The tolerance (TL) is 0, 1, 2 respectively.



Graph 2: Avg. leaf accesses per query with and without forced-reinsert (solid and dashed lines, respectively). The tolerance is 0,1.

on clustering methods, e.g., in Information Retrieval, mainly focuses on nearest neighbor queries that are *partially* specified, using ''recall'' and ''precision'' as performance measures, and rarely reporting disk accesses.



Graph 3: Avg. leaf accesses per query, as a function of the tolerance *TL*. *N*=1, 2, and 3 thousand items.

	tol=0	tol=1	tol=2
N=1000	0.0101	0.0250	0.0490
N=2000	0.0064	0.0172	0.0406
N=3000	0.0059	0.0144	0.0353

Table 1: Average fraction of leaf nodes touched per query

Few papers report measurements or estimates of the search effort for high-dimensional spaces. They typically measure the proportion of records accessed during a nearest neighbor query: Shasha and Wang [33] report accessing ≈20%-80% of the database; Friedman et al. [10] report that their method tends to access almost 100% of the database, for dimensionalities ≈8-9 and above. Some more details about both these papers are presented in the next section.

For comparison, we computed the average fraction of leaf nodes that are touched per query. Table 1 shows the results, as a function of the number of elements *N* in the database and the tolerance of the query. A direct comparison of the Diamond-tree with the two previous methods is not very meaningful, because the environments are very different. In any case, the Diamond-tree touches <1% for exact match queries, <2.5% for queries with tolerance=1, and <5% for queries with tolerance=2.

In conclusion, the search effort of the Diamond tree is very promising, not only as a percentage of the sequential scan effort, but as an absolute value, as well. Moreover, it should be noted that our implementation had several shortcuts, aiming mainly to prove the feasibility of the method. Future versions of the Diamond-tree will benefit from several performance boosters, such as (a) dimensionality reduction algorithms, (b) more careful thresholding, (c) the exact MBD algorithm, as opposed to the approximate one etc.

5. RELATED WORK.

Most nearest-neighbor searching methods work well for low-dimensionality spaces. We present a brief survey here and explain why they do not work well for spaces with many dimensions. Nearest neighbor searching techniques can be classified in two groups. The first group exploits the fact that the points are embedded in a co-ordinate space. The second exploits the triangular inequality.

In the first group fall the k-d tree inspired methods: Friedman et al. [11] use a k-d tree on the co-ordinate space, to locate the nearest neighbor in logarithmic time. Bentley and Weide [5] divide the plane in a square grid and find the nearest neighbors in constant expected time, if the points are drawn from a uniform distribution. They use a "spiral search", ie., examine the cell of the query point, then the cells that are one step away, then two steps away etc., until the closest match is found. The method could be generalized for n-d co-ordinate spaces, but the complexity of the spiral search grows exponentially with n. Friedman et al. [10] project the points on one dimension, choose the nearest neighbors on this dimension, and examine them until the actual nearest neighbor is found. The method works well for low dimensionality ($d \leq 8$), as the authors mention. For higher dimensionalities, almost the whole file has to be examined. Eastman and Zemankova [9] derive a formula for the search performance of fully and partially specified nearest-neighbor queries.

There are many other methods for spatial data organizations, based on hierarchical space decomposition, suitable for disk-based storage: k-d B-trees [28]; are a direct extension of k-d trees. The R-tree [13] family of methods allow overlapping parent nodes with several variants: R+ trees [32], packed R-trees [29], R*-trees [4], and P-trees [18]; Another family consists of methods that allow for holes in the parent nodes [23]. A closely related family of methods is based on a flat, grid-based, decomposition of the space, with the grid file as the main representative [25] and several variants, e.g. [16].

All these methods do not work well for high-dimensionality sparse vectors: the k-d tree method will lead to highly un-balanced trees, because most of the attributes have a 0 value; the R-tree based methods will have a low fan-out, because of the large space required to store each vector/rectangle; the grid-file methods will require too much space for the directory, which grows exponentially with the dimensionality.

The second group of methods, based on the triangular inequality, include the following: Ito and Kizawa [17] suggest imposing a linear ordering on the objects, by traversing their minimum spanning tree. However, their method seems fine-tuned for words of a dictionary and specifically for typing errors. Fukunaga and Narendra [12] proposed the "branch and bound" algorithm we adopted, to search a cluster hierarchy for nearest neighbors. However, they do not consider insertions and deletions, nor efficient methods of building the tree structure. Shasha and Wang [33] precompute and store the distances from several pairs of points. Using the triangle inequality, they can infer the lower and upper bound on every other distance of interest; the results are stored in a $n \times n$ matrix, called the ADM (=Approximate Distance Map); n is the number of points. According to the graphs they reported, their method examines ≈20%-80% of the n points. The computation of the ADM requires $O(n^3)$ steps, which becomes prohibitive for large databases.

Relevant to our work is a wide variety of clustering algorithms (see, for example, [24, 30, 31, 34] for surveys). However, their main goals are to detect patterns in the data; there is little attention paid to the time required for cluster creation and update operations, especially for large databases.

6. CONCLUSIONS

We have examined the problem of indexing points in a high-dimensionality space, to handle efficiently fully specified approximate searches. We have designed and implemented the Diamond-tree structure, which exhibits the following virtues:

- It handles high as well as low dimensionalities; it could even be used for 1-d spaces, replacing the B-tree. Thus, it could be used as a generic method for indexing in Object-Oriented databases, [1, 6, 14] to handle several data types, such as strings, images, fingerprints etc.
- It can even handle the case of a *dynamically changing* feature set, as long as there are enough unused feature-ids.
- It is designed to work on disks, to support large databases
- It guarantees 50% space utilization, and it is implemented as a balanced tree.

Preliminary experimentation shows very good performance on a 3,000 item database (words, searched for typing errors). The method touches ≈1%-5% of the data, in a 26-dimensional space; related methods for approximate searching touch much larger portions (e.g., 20%-80% of the size of the file in [33], or almost 100% of the file, for high-dimensionality spaces [10]). Moreover, in the proposed method, the absolute number of leaves touched is small, promising response times below 1 second.

The major ideas and contributions in this work are

- The use of compressed vectors to handle high-dimensionality points; the use of diamonds (as opposed to rectangles) to save space and increase the fanout of the nodes of the tree
- Manipulation algorithms for the diamonds (calculation of minimum bounding diamond, splitting etc.)
- Implementation and experimental results, that show that the method is feasible and promising for real applications.

ACKNOWLEDGMENTS

The authors would like to thank Ken Clarkson and Steve Fortune for their help with the formulation of the linear programming problem for the minimum bounding diamond (MBD).

References

1. Agrawal, R. and N.H. Gehani, "Ode (Object Database and Environment): The Language and the Data Model," *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, pp. 36-45, Portland, Oregon, May-June 1989.
2. Altschul, S.F., W. Gish, W. Miller, E.W. Myers, and D.J. Lipman, "A Basic Local Alignment Search Tool," *Journal of Molecular Biology*, 1990.
3. Angell, R.C., G.E. Freund, and P. Willet, "Automatic Spelling Correction Using a Trigram Similarity Measure," *Information Processing and Management*, vol. 19, no. 4, pp. 255-261, 1983.
4. Beckmann, N., H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles," *ACM SIGMOD*, pp. 322-331, Atlantic City, NJ, May 23-25, 1990.
5. Bentley, Jon Louis, Bruce W. Weide, and Andrew C. Yao, "Optimal Expected-Time Algorithms for Closest Point Problems," *ACM Trans. on Mathematical Software (TOMS)*, vol. 6, no. 4, pp. 563-580, Dec. 1980.
6. Carey, M.J., D.J. DeWitt, G. Graefe, D.M. Haight, J.E. Richardson, D.H. Schuh, E.J. Shekita, and S.L. Vandenberg, "The EXODUS Extensible DBMS Project: An Overview," in *Readings in Object-Oriented Database Systems*, ed. D. Maier, Morgan Kaufmann, 1990.
7. Deerwester, S., S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by Latent Semantic Analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391-407, Sept. 1990.
8. Durham, Ivor, David A. Lamb, and James B. Saxe, "Spelling Correction in User Interfaces," *CACM*, Oct. 1983.
9. Eastman, C.M. and Maria Zemankova, "Partially Specified Nearest Neighbor Searches Using k-d Trees," *Information Processing Letters (IPL)*, vol. 15, no. 2, pp. 53-56, Sept. 1982.
10. Friedman, Jerome H., Forest Baskett, and Leonard H. Shustek, "An Algorithm for Finding Nearest Neighbors," *IEEE Trans. on Computers (TOC)*, vol. C-24, pp. 1000-1006, Oct. 1975.
11. Friedman, Jerome H., Jon Louis Bentley, and R.A. Finkel, "An Algorithm for Finding Best Matches in Logarithmic Expected Time," *ACM Trans. on Math. Software (TOMS)*, vol. 3, no. 3, pp. 209-226, Sept. 1977.
12. Fukunaga, Keinosuke and Patrenahalli M. Narendra, "A Branch and Bound Algorithm for Computing k-Nearest Neighbors," *IEEE Trans. on Computers (TOC)*, vol. C-24, no. 7, pp.

750-753, July 1975.

13. Guttman, A., ''R-Trees: A Dynamic Index Structure for Spatial Searching,'' *Proc. ACM SIGMOD*, pp. 47-57, Boston, Mass, June 1984.

14. Haas, L. M., J. C. Freytag, G. M. Lohman, and H. Pirahesh, ''Extensible Query Processing in Starburst,'' *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data*, pp. 377-388, Portland, Oregon, May-June 1989.

15. Hollaar, L.A., ''Text Retrieval Computers,'' *IEEE Computer Magazine*, vol. 12, no. 3, pp. 40-50, March 1979.

16. Hutflesz, A., H.-W. Six, and P. Widmayer, ''Twin Grid Files: Space Optimizing Access Schemes,'' *Proc. of ACM SIGMOD*, pp. 183-190, Chicago, Illinois, June 1-3, 1988.

17. Ito, Tetsuro and Makoto Kizawa, ''Hierarchical File Organization and Its Applications to Similar-String Matching,'' *ACM TODS*, vol. 8, no. 3, pp. 410-433, Sept. 1983.

18. Jagadish, H. V., ''Spatial Search with Polyhedra,'' *Proc. Sixth IEEE Int'l Conf. on Data Engineering*, Los Angeles, CA, Feb 1990.

19. Jagadish, H.V., ''A Retrieval Technique for Similar Shapes,'' *Proc. ACM SIGMOD Conf.*, pp. 208-217, Denver, Colorado, May 29-31, 1991.

20. Jain, R. and W. Niblack, *NSF Workshop on Visual Information Management*, Redwood City, CA, Feb. 1992.

21. Johnson, J.H., ''Formal Models for String Similarity,'' Research Report CS-83-32, Univ. of Waterloo, Waterloo, Ontario, Canada, Nov. 1983. Ph.D. Dissertation

22. Jones, Mark A., Guy A. Story, and Bruce W. Ballard, ''Integrating Multiple Knowledge Sources in a Bayesian OCR Post-Processor,'' *First International Conference on Document Analysis and Recognition*, Saint-Malo, France, Sept. 1991. to appear

23. Lomet, David B. and Betty Salzberg, ''The hB-tree: A Multiattribute Indexing Method with Good Guaranteed Performance,'' *ACM TODS*, vol. 15, no. 4, pp. 625-658, Dec. 1990.

24. Murtagh, F., ''A Survey of Recent Advances in Hierarchical Clustering Algorithms,'' *The Computer Journal*, vol. 26, no. 4, pp. 354-359, 1983.

25. Nievergelt, J., H. Hinterberger, and K.C. Sevcik, ''The Grid File: An Adaptable, Symmetric Multikey File Structure,'' *ACM TODS*, vol. 9, no. 1, pp. 38-71, March 1984.

26. O'Gorman, Lawrence and Jeffrey V. Nickerson, ''An approach to fingerprint filter design,'' *Pattern Recognition*, pp. 29-38, Jan. 1989.

27. Peterson, J.L., ''Computer Programs for Detecting and Correcting Spelling Errors,'' *CACM*, vol. 23, no. 12, pp. 676-687, Dec. 1980.

28. Robinson, J.T., ''The k-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes,'' *Proc. ACM SIGMOD*, pp. 10-18, 1981.

29. Roussopoulos, N. and D. Leifker, ''Direct Spatial Search on Pictorial Databases Using Packed R-Trees,'' *Proc. ACM SIGMOD*, Austin, Texas, May 1985.

30. Salton, G. and M.J. McGill, *Introduction to Modern Information Retrieval,* McGraw-Hill, 1983.

31. Salton, G. and A. Wong, ''Generation and Search of Clustered Files,'' *ACM TODS*, vol. 3, no. 4, pp. 321-346, Dec. 1978.

32. Sellis, T., N. Roussopoulos, and C. Faloutsos, ''The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects,'' *Proc. 13th International Conference on VLDB*, pp. 507-518, England,, Sept. 1987. also available as SRC-TR-87-32, UMIACS-TR-87-3, CS-TR-1795

33. Shasha, Dennis and Tsong-Li Wang, ''New Techniques for Best-Match Retrieval,'' *ACM TOIS*, vol. 8, no. 2, pp. 140-158, Apr. 1990.

34. Van-Rijsbergen, C.J., *Information Retrieval,* Butterworths, London, England, 1975. 1st edition

35. Wallace, Gregory K., ''The JPEG Still Picture Compression Standard,'' *CACM*, vol. 34, no. 4, pp. 31-44, Apr. 1991.