Similarity Search: A Matching Based Approach

Anthony K. H. Tung[†] Rui Zhang[‡] Nick Koudas[§] Beng Chin Ooi[†]

[†] National Univ. of Singapore
{atung, ooibc}@comp.nus.edu.sg

[‡] Univ. of Melbourne
rui@csse.unimelb.edu.au

[§] Univ. of Toronto
koudas@cs.toronto.edu

ABSTRACT

Similarity search is a crucial task in multimedia retrieval and data mining. Most existing work has modelled this problem as the nearest neighbor (NN) problem, which considers the distance between the query object and the data objects over a *fixed* set of features. Such an approach has two drawbacks: 1) it leaves many partial similarities uncovered; 2) the distance is often affected by a few dimensions with high dissimilarity. To overcome these drawbacks, we propose the *k-n-match* problem in this paper.

The k-n-match problem models similarity search as matching between the query object and the data objects in n dimensions, where n is a given integer smaller than dimensionality d and these n dimensions are determined dynamically to make the query object and the data objects returned in the answer set match best. The k-n-match query is expected to be superior to the kNN query in discovering *partial similarities*, however, it may not be as good in identifying *full similarity* since a single value of n may only correspond to a particular aspect of an object instead of the entirety. To address this problem, we further introduce the *frequent k-n-match* problem, which finds a set of objects that appears in the k-n-match answers most frequently for a range of n values. Moreover, we propose search algorithms for both problems. We prove that our proposed algorithm is optimal in terms of the number of individual attributes retrieved, which is especially useful for information retrieval from multiple systems. We can also apply the proposed algorithmic strategy to achieve a disk based algorithm for the (frequent) k-n-match query. By a thorough experimental study using both real and synthetic data sets, we show that: 1) the k-n-match query yields better result than the kNN query in identifying similar objects by partial similarities; 2) our proposed method (for processing the frequent k-n-match query) outperforms existing techniques for similarity search in terms of both effectiveness and efficiency.

VLDB '06, September 12-15, 2006, Seoul, Korea.

1. INTRODUCTION

Similarity search is a crucial task in many multimedia and data mining applications and extensive studies have been performed in the area. Usually, the objects are mapped to multi-dimensional points and similarity search is modelled as a nearest neighbor search in a multi-dimensional space. In such searches, comparison between two objects is performed by computing a score based on a similarity function like Euclidean distance [8] which essentially aggregates the difference between each dimension of the two objects. The nearest neighbor model considers the distance between the query object and the data objects over a *fixed* set of features. Such an approach has two drawbacks: 1) it leaves many partial similarities uncovered since the distance computation is based on the fixed set of features; 2) the distance is often affected by a few dimensions with high dissimilarity. For example, consider the 10-dimensional database consisting of four data objects as shown in Figure 1 and the query object (1,1,1,1,1,1,1,1,1,1). A search for the nearest neighbor based on Euclidean distance will return object 4 as the answer. However, it is not difficult to see that the other three objects are actually more similar to the query object in 9 out of the 10 dimensions but are considered to be further away due to large dissimilarity in only one dimension (the dimension with value 100). Such a phenomenon becomes more obvious for high-dimensional data when the probability of encountering big differences in some of the dimensions is higher. These high dissimilarity dimensions happen often in real applications such as bad pixels, wrong readings or noise in a signal. Moreover, think of the example database as features extracted from pictures, and suppose the first three dimensions represent the color, the second three dimensions represent the texture and the last four dimensions represent the shape (there may be more number of features in reality). We can see that the nearest neighbor based on Euclidean distance returns a picture which is not that similar to the query picture in any aspects despite those pictures

ID	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}
1	1.1	100	1.2	1.6	1.6	1.1	1.2	1.2	1	1
2	1.4	1.4	1.4	1.5	100	1.4	1.2	1.2	1	1
3	1	1	1	1	1	1	2	100	2	2
4	20	20	20	20	20	20	20	20	20	20

Figure 1: An Example Database

that have exact matches on certain aspects (e.g., picture 3 matches the query's color and texture exactly). This shows how finding similarity based on a fixed set of features overlook the partial similarities.

To overcome these drawbacks, we propose the *k-n-match* problem in this paper. For ease of illustration, we will start with the *n-match* problem, which is the special case when k equals 1. Alternatively, we can view the *k-n-match* problem as finding the top k answers for the *n-match* problem.

The n-match problem models similarity search as matching between the query object and the data objects in n dimensions, where n is a given integer smaller than dimensionality d and these n dimensions are determined dynamically to make the query object and the data objects returned in the answer set match best. A key difference here is that we are using a small number (n) of dimensions which are determined dynamically according to the query object and a particular data object, so that we can focus on the dimensions where these two objects are most similar. By this means, we overcome drawback (2), that is, the effects of the dimensions with high dissimilarities are suppressed. Further, by using n dimensions, we are able to discover partial similarities so that drawback (1) is overcome. To further give an intuition on the method that we are proposing in this paper, let us consider the process of judging the similarity between two persons. Given the large number of features (eye color, shape of face etc.) and the inability to give a very accurate measure of the similarity for each feature, a quick approach is to approximate the number of features in which we judge to be very close and claim that the two persons are similar if the number of such features is high. Still consider the example in Figure 1, if we issue a 6-match query, object 3 will be returned, which is a more reasonable answer than object 4. However, we may not always have exact matches in reality especially for continuous value domains. For example, objects 1 and 2 are also close to the query in many dimensions although they are not exact matches. Therefore, we use a more flexible match scheme, that is, p_i (data value in dimension i) matches q_i (query value in dimension i) if their difference is within a threshold δ. If we set δ to 0.2, we would have an additional answer, object 1, for the 6-match query. A new problem here is how we determine δ. We still leave this choice self-adaptive with regard to the data and query. Specifically, for a data object P and a query Q, we first sort the differences $|p_i - q_i|$ in all the dimensions and obtain the n-th smallest difference, called P's **n-match difference** with regard to Q. Then, among all the data objects, the one with the smallest n-match difference determines δ, that is, δ is this smallest n-match difference. For the k-n-match query, δ equals the k-th smallest n-match difference and therefore k objects will be returned as the answer.

While the k-n-match query is expected to be superior than the kNN query in discovering partial similarities, it may not be as good in finding *full similarity* since a single value of n may only correspond to one aspect of an object instead of the entirety. To address the problem, we further introduce the *frequent k-n-match* query. In the frequent k-n-match query, we first find out the k-n-match solutions for a range of n values, say, from 1 to d. Then we choose k objects that appear most frequently in the k-n-match answer sets for all the n values.

A naive algorithm for processing the k-n-match query is to compute the n-match difference of every point and return the top k answers. The frequent k-n-match query can be done similarly. We just need to maintain a top k answer set for each n value required by the query while checking every point. However, the naive algorithm is expensive since we have to scan the whole database and hence every attribute of every point is accessed. In this paper we propose an algorithm that works on a different organization of the data objects, namely, each dimension of the data set is sorted. Our algorithm accesses the attributes in each dimension in ascending order of their differences to the query in corresponding dimensions. We call it the **ascending difference** (**AD**) algorithm. We prove that the AD algorithm is optimal for both query types in terms of the number of individual attributes retrieved given our data organization. Our model of organizing data as sorted dimensions and using number of attributes retrieved as cost measure matches very well the setting of information retrieval from multiple systems [11]. Our cost measure also conforms with the cost model of disk based algorithms, where the number of disk accesses is the major measure of performance, and the number of disk accesses is proportional to the attributes retrieved. So we also apply our algorithmic strategy to achieve an efficient disk based algorithm.

By a thorough experimental study using both real and synthetic data sets, we show that: 1) the k-n-match query yields better result than the kNN query in identifying similar objects by partial similarities; 2) our proposed method (for processing the frequent k-n-match query) outperforms existing techniques for similarity search in terms of both effectiveness and efficiency.

The rest of the paper is organized as follows. First, we formulate the k-n-match and the frequent k-n-match problems in Section 2. Then we propose the AD algorithm for processing the (frequent) k-n-match problem and discuss properties of the AD algorithm in Section 3. In section 4, we apply our algorithmic strategy to achieve a disk based solution. At the same time, we give an adapted algorithm from the VA-file technique as a competitive method for the disk based version of the problem. The experimental study is reported in Section 5 and related work is discuss in Section 6. Finally, we conclude the paper in Section 7.

2. PROBLEM FORMULATION

In this section, we formulate the k-n-match and the frequent k-n-match problems. As an object is represented as a multi-dimensional point, we will use *object* and *point* interchangeably in the remainder of the paper. Then a database is a set of d-dimensional points, where d is the dimensionality. The notation used in this paper is summarized in Table 1 for easy reference.

2.1 The K-N-Match Problem

For ease of illustration, we start with the simplest form of the k-n-match problem, that is, the n-match problem, which is the special case when k equals 1. Before giving the definition, we first define the n-match difference of a point P with regard to another point Q as follows:

Table 1: Notation

Notation	Meaning
c	Cardinality of the database
DB	The database, which is a set of points
d	Dimensionality of the data space
k	The number of n-match points to return
n	The number of dimensions to match
P	A point
p_i	The coordinate of P in the i-th dimension
Q	The query point
q_i	The coordinate of Q in the i-th dimension
S	A set of points

Definition 1. ***N*-match difference**
Given two d-dimensional points P $(p_1, p_2, ..., p_d)$ and Q $(q_1, q_2, ..., q_d)$, let $\delta_i = |p_i - q_i|$, $i = 1, ..., d$. Sort the array $\{\delta_1, ..., \delta_d\}$ in increasing order and let the sorted array be $\{\delta'_1, ..., \delta'_d\}$. Then δ'_n is the n-match difference of point P with regard to Q. □

Following our notation, Q represents the query point, therefore in the sequel, we simply say P's n-match difference for short and by default it is with regard to the query point Q. Obviously, the n-match difference of point P with regard to Q is the same as the n-match difference of point Q with regard to P.

Next, we give the definition of the n-match problem as follows:

Definition 2. **The *n*-match problem**
Given a d-dimensional database DB, a query point Q and an integer n $(1 \leq n \leq d)$, find the point $P \in DB$ that has the smallest n-match difference with regard to Q. P is called the n-match point of Q. □

In the example of Figure 1, point 3 is the 6-match (δ=0) of the query, point 1 is the 7-match (δ=0.2) and point 2 is the 8-match (δ=0.4).

Figure 2 shows a more intuitive example in 2-dimensional space. A is the 1-match of Q because it has the smallest difference from Q in dimension x. B is the 2-match of Q because when we consider 2 dimensions, B has the smallest difference.

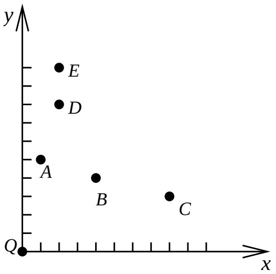


Figure 2: The n-match problem

Intuitively, the query finds a point P that matches Q in n dimensions. If we consider the maximum difference in these n dimensions, P has the smallest maximum difference to Q among all the points in DB.

This definition conforms with our reasoning in the example of Figure 1, which actually uses a modified form of Hamming distance [15] in judging the similarity exhibited by the first three points. The difference however is that we are working on spatial attributes while Hamming distance is typically used for categorical data [1].

If we view n-match difference as a distance function, we can see that the n-match problem is still looking for the nearest neighbor of Q. The key difference is that the distance is not defined on a fixed set of dimensions, but dynamically determined based on the query and data points. The n-match difference differs from existing similarity scores in two ways. First, the attributes are discretized dynamically by determining a value of δ on the fly. Given a value of δ, determining a match or a mismatch is performed independently without aggregating the actual differences among the dimensions. Because of this, dimensions with high dissimilarities are not accumulated, making comparison more robust to these artifacts. Second, in the n-match problem, the number of dimensions that are deemed close are captured in the final result. Existing similarity measure can generally be classified into two approaches. For categorical data, the total number of dimensions in which there are matches are usually used. For spatial data, a distance function like Euclidean distance simply aggregates the differences without capturing any dimensional matching information. The approach of n-match can be seen as combination of the two, capturing the number of dimensional matches in terms of n and the spatial distances in terms of δ. This makes sense especially in high dimensional data in which we can leverage on a high value of d to provide statistical evidence that two points are similar if they are deemed to be close in most of the d dimensions.

Note that our distance function is not a generalization of the Chebyshev distance (or the L_∞ norm), which returns the maximum difference (made to positive) of attributes in the same dimension. The radical difference is that our function is **not** metric, particularly, it does not satisfy the triangular inequality. Consider the three 3-dimensional points F(0.1, 0.5, 0.9), G(0.1, 0.1, 0.1) and H(0.5, 0.5, 0.5). The 1-match difference between F and G, F and H, G and H are 0, 0, 0.4, respectively; they do not satisfy the triangular inequality of $0 + 0 > 0.4$.

In analogy to kNN with regard to NN, we further introduce the k-n-match problem as follows.

Definition 3. **The *k*-*n*-match problem**
Given a d-dimensional database DB of cardinality c, a query point Q, an integer n $(1 \leq n \leq d)$, and an integer $k \leq c$, find a set S which consists of k points from DB so that for any point $P1 \in S$ and any point $P2 \in DB - S$, the n-match difference between P1 and Q is less than or equal to the n-match difference between P2 and Q. The S is the k-n-match set of Q.

For the example in Figure 2, $\{A, D, E\}$ is the 3-1-match of Q while $\{A, B\}$ is the 2-2-match of Q.

Obviously the k-n-match query is different from the skyline query, which returns a set points so that any point in

[1] A side effect of our work will be that we can have a uniform treatment for both type of attributes in the future.

the returned set is not dominated by any other point in the database. The skyline query returns $\{A, B, C\}$ for the example in Figure 2, while the k-n-match query returns k points depending on the query point and the k value. None of the k-n-match query example shown above has the same answer as the skyline query.

While the k-n-match problem may find us similar objects through partial similarity, the choice of n introduces an additional parameter to the solution. It is evident that the most similar points returned are sensitive to the choice of n. To address this, we will further introduce the *frequent k-n-match query*, which is described in the following section.

2.2 The Frequent K-N-Match Problem

The k-n-match query can help us find out similar objects through partial similarity when an appropriate value of n is selected. However, it is not obvious how such a value of n can be determined. Instead of trying to find such a value of n directly, we will instead vary n within a certain range (say, 1 to d) and try to compute some statistics on the set of matches that are returned for each n. Specifically, we first find out the k-n-match answer sets for a range $[n_0, n_1]$ of n values. Then we choose the k points that appear most frequently in the k-n-match answer sets for all the n values. Henceforth, we will say that the similar points generated from the k-n-match problem are based on *partial similarity* (only one value of n) while those generated from the frequent k-n-match problem are based on *full similarity* (all possible values of n). We use an example to illustrate the intuition behind such a definition. Suppose we are looking for objects similar to an orange. The objects are all represented by its features including color (described by 1 attribute), shape (described by 2 attributes) and other characteristics. When we issue a k-1-match query, we may get a fire and a sun in the answer set. When we issue a k-2-match query, we may get a volleyball and a sun in the answer set. The sun appears in both answer sets while none of the volleyball or the fire does, because the sun is more similar to the orange than the others, in both color and shape.

The definition of the frequent k-n-match problem is given below:

Definition 4. ***The frequent k-n-match problem***
Given a d-dimensional database DB of cardinality c, a query point Q, an integer $k \leq c$, and an integer range $[n_0, n_1]$ within $[1, d]$, let S_0, ..., S_i be the answer sets of k-n_0-match, ..., k-n_1-match, respectively. Find a set T of k points, so that for any point $P_1 \in T$ and any point $P_2 \in DB - T$, P_1's number of appearances in S_0, ..., S_i is larger than or equal to P_2's number of appearances in S_0, ..., S_i.

The range $[n_0, n_1]$ can be determined by users. We can simply set it as $[1, d]$. As in our previous discussion, full number of dimensions usually contains dimensions of large dissimilarity, therefore setting n_1 as d may not help much in the effectiveness. On the other hand, too few features can hardly determine a certain aspects of an object and matching on a small number of dimensions may be caused by noises. Therefore, we may set n_0 as a small number, say 3, instead of 1. We will investigate more on the effects of n_0 and n_1 in Section 5 through experiments.

3. ALGORITHMS

In this section, we propose an algorithm to process the (frequent) k-n-match problem with optimal cost under the following model, namely, attributes are sorted in each dimension and the cost is measured by the number of attributes retrieved. This model makes sense in a number of settings. For example, in information retrieval from multiple systems [11], objects are stored in different systems and given scores by each system. Each system will sort the objects according to their scores. A query retrieves the scores of objects (by sorted access) from different systems and then combines them using an aggregation function to obtain the final result. In this whole process, the major cost is the retrieval of the scores from the systems, which is proportional to the number of scores retrieved. [11] has focused on aggregation functions such as *min* and *max*. Besides these functions, we could also perform similarity search over the systems and implement similarity search as the (frequent) k-n-match query. Then the scores from different systems become the attributes of different dimensions in the (frequent) k-n-match problem, and the algorithmic goal is to minimize the number of attributes retrieved. Further, the cost measure also conforms with the cost model of disk based algorithms, where the number of disk accesses is the major measure of performance, and the number of disk accesses is proportional to the attributes retrieved. However, unlike the multiple system information retrieval case, disk based schemes may make use of indexes to reduce disk accesses, which adds some complexity to judge which strategy is better. We will analyze these problems in more detail in Section 4.

A naive algorithm for processing the k-n-match query is to compute the n-match difference of every point and return the top k answers. The frequent k-n-match query can be done similarly. We just need to maintain a top k answer set for each n value required by the query while checking every point. However, the naive algorithm is expensive since every attribute of every point is retrieved. We hope to do better and access less than all the attributes. We will propose an algorithm, called the AD algorithm, that retrieves minimum number of attributes in Section 3.1.

Note that the algorithm proposed in [11] for aggregating scores from multiple systems, called *FA*, does not apply to our problem. They require the aggregation function to be monotone, but the aggregation function used in k-n-match (that is, n-match difference) is not monotone. We use an example to explain this. Consider the database in Figure 3 and we are looking for the 1-match of the query $(3.0, 7.0, 4.0)$. A function f is monotone means that $f(p_1, ..., p_d) \leq f(p'_1, ..., p'_d)$ whenever $p_i \leq p'_i$ for every $i = 1, ..., d$ (or $p_i \geq p'_i$ for every

ID	d_1	d_2	d_3
1	0.4	1.0	1.0
2	2.8	5.5	2.0
3	6.5	7.8	5.0
4	9.0	9.0	9.0
5	3.5	1.5	8.0

Figure 3: An Example Database

$i = 1, ..., d$). In the example, point 1 is smaller than point 2 in every dimension, but its 1-match difference (2.6) is larger than point 2's 1-match difference (0.2). Point 4 is larger than point 2 in every dimension, but its 1-match difference (2.0) is still larger than point 2's 1-match difference (0.2). This example shows that the n-match difference is not a monotone aggregation function. If we use the FA algorithm here, we get point 1, which is a wrong answer (the correct answer is point 2). The reason is that the sorting of the attributes in each dimension is based on the attribute values, but our ranking is based on the differences to the query. Furthermore, the score we obtained from the aggregation function (n-match difference) is based on a dynamically determined dimension set instead of all the dimensions.

Next we present the AD algorithm, which guarantees correctness of the answer and retrieves minimum number of attributes.

3.1 The AD Algorithm for K-N-Match Search

Recall the model that the attributes are sorted in each dimension; each attribute is associated with its point ID. Therefore, we have d sorted lists. Our algorithm works as follows. We first locate each dimension of the query Q in the d sorted lists. Then we retrieve the individual attributes in ascending order of their differences to the corresponding attributes of Q. When a point ID is first seen n times, this point is the first n-match. We keep retrieving the attributes until k point ID's have been seen at least n times. Then we can stop. We call this strategy of accessing the attributes in *Ascending* order of their *Differences* to the query point's attributes as the **AD** algorithm. Besides the applicability due to the aggregation function, the AD algorithm has another key difference from the FA algorithm in the accessing style. The FA algorithm accesses the attributes in parallel, that is, if we think of the sorted dimensions as columns and combine them into a table, the FA algorithm would access the "records" in the table one row after another. But the AD algorithm access the attributes in ascending order of their differences to the corresponding query attributes. If a parallel access was used, we would retrieve more attributes than necessary as can be seen from the optimality analysis in Theorem 3.2.

The detailed steps of the AD algorithm for k-n-match search, namely "KNMatchAD", is illustrated in Figure 4. Line 1 initializes some structures used in the algorithm. $appear[i]$ maintains the number of appearances of point i. It has c elements, where c is the cardinality of the database[2], and all the elements are initialized to 0. h maintains the number of point ID's that have appeared n times and is initialized to 0. S is the answer set and initialized to $\emptyset$. Line 3 finds the position of q_i in dimension i using a binary search, since each dimension is sorted. Then starting from the position of q_i, we access the attributes one by one towards both directions along dimension i. Here, we use an array $g[\ ]$ (line 4) of size $2d$ to maintain the next attribute to access in each dimension, in both directions (attributes smaller than q_i and attributes larger than q_i). Actually we can view them as $2d$ dimensions: the direction towards smaller values of dimension i corresponds to $g[2*(i-1)]$ while the direction towards large values of dimension i corresponds to $g[2*i-1]$. Each element of $g[\ ]$ is a triple (pid, pd, dif) where pid is the point ID of the attribute, pd is the dimension and dif is the difference between q_{pd} and the next attribute to access in dimension pd. For example, first we retrieve the largest attribute in dimension 1 that is smaller than q_0, let it be a_0 and let its point ID be pid_0. We use them to form the triple $(pid_0, 0, q_0 - a_0)$, and put this triple into $g[0]$. Similarly, we retrieve the smallest attribute in dimension 1 that is larger than q_0 and form a triple to be put into $g[1]$. We do the same thing for other dimensions. After initializing $g[\ ]$, we begin to pop out values from it in the ascending order of dif. The function "smallest" in line 6 returns the triple with the smallest dif from $g[\ ]$. Whenever we see a pid, we increase its number of appearance by 1 (line 7). When a pid appears n times, an n-match is found, therefore h is increased by 1 and the pid is added to S. After popping out an attribute from $g[\ ]$, we retrieve the next attribute in the same dimension to fill the slot. Next, we continue to pop out triples from $g[\ ]$ until h reaches k, and then the algorithm terminates.

```
Algorithm KNMatchAD
1  Initialize $appear[\ ]$, $h$ and $S$.
2  for every dimension $i$
3      Locate $q_i$ in dimension $i$.
4      Calculate the differences between $q_i$ and its
       closest attributes in dimension $i$ along both
       directions. Form a triple $(pid, pd, dif)$ for each
       direction. Put this triple to $g[pd]$.
5  do
6      $(pid, pd, dif) = \text{smallest}(g)$;
7      $appear[pid]$++;
8      if $appear[pid] = n$
9          $h$++;
10         $S = S \cup pid$;
11     Read next attribute from dimension $pd$ and form
       a new triple $(pid, pd, dif)$. If end of the dimension
       is reached, let $dif$ be $\infty$. Put the triple to $g[pd]$.
   while $h < k$
12 return $S$.
End KNMatchAD
```

Figure 4: Algorithm KNMatchAD

We use the database in Figure 3 as a running example to explain the algorithm, and suppose we are searching 2-2-match for the query $(3.0, 7.0, 4.0)$. Hence k=n=2 in this query. First, we have each dimension sorted as in Figure 5, where each entry in each dimension represents a (point ID, attribute) pair. We locate q_i in each dimension. q_1 is between (2, 2.8) and (5, 3.5); q_2 is between (2, 5.5) and (3, 7.8); q_3 is between (2, 2.0) and (3, 5.0). Then we calculate the differences of these attributes to q_i in the corresponding dimension and form triples, which are put into the array $g[\ ]$. $g[\ ]$ becomes {(2, 0, 0.2), (5, 1, 0.5), (2, 2, 1.5), (3, 3, 0.8), (2, 4, 2.0), (3, 5, 1.0)}. Then we start popping triples out of $g[\ ]$ from the one with the smallest difference. First, (2, 0,

[2] We only use 1 byte for each element of $appear[\ ]$, which can work for up to 256 dimensions. For a data set of 1 million records, the memory usage is 1 Megabytes. This should be acceptable given the memory size of today's computer.

d_1	d_2	d_3
1, 0.4	1, 1.0	1, 1.0
2, 2.8	5, 1.5	2, 2.0
5, 3.5	2, 5.5	3, 5.0
3, 6.5	3, 7.8	5, 8.0
4, 9.0	4, 9.0	4, 9.0

Figure 5: A Running Example

0.2) is popped out, so *appear*[2] is increased by 1 and equals 1 now. We read the next pair in dimension 1 towards the smaller attribute direction, that is, (1, 0.4), and form the triple (1, 0, 2.6), which is put back into $g[0]$. Next, we pop the triple with the smallest difference from the current $g[\ ]$. We get (5, 1, 0.5), so *appear*[5] becomes 1 and (3, 1, 3.5) is put into $g[1]$. Next, we get (3, 3, 0.8) from $g[\ ]$, so *appear*[3] becomes 1 and (4, 3, 2.0) is put into $g[3]$. Now $g[\ ]$={(1, 0, 2.6), (3, 1, 3.5), (2, 2, 1.5), (4, 3, 2.0), (2, 4, 2.0), (3, 5, 1.0)}. Next, we get (3, 5, 1.0), so *appear*[3] becomes 2, which equals n, and so h becomes 1. At this time we have found the first 2-match point, that is, point 3. (5, 5, 4.0) is put into $g[5]$. Next we get (2, 2, 1.5) from $g[\ ]$, so *appear*[2] becomes 2, which also equals n, and so h becomes 2, which equals k. At this time we have found the second 2-match, therefore the algorithm stops. The 2-2-match set is {point 2, point 3} and we also get the 2-2-match difference, 1.5.

In the implementation, we do not have to actually store pd in the triple since we can tell which dimension a (pid, dif) pair is from when we get it from the sorted dimensions or from $g[\ ]$.

In what follows, we will prove the correctness and optimality of the AD algorithm.

Theorem 3.1. ***Correctness of KNMatchAD***
The points returned by algorithm KNMatchAD compose the k-n-match set of Q.

Proof. We will prove that the k-th point that appears n times has the k-th smallest n-match difference.

First, we consider $k = 1$. Let the first point that appears n times be P, and when it appears the n-th time, let the difference between p_i and q_i be dif (i is the corresponding dimension). We are accessing the attributes in ascending order of their differences to q_i, therefore dif is the n-match difference of P. Suppose P does not have the smallest n-match difference, then there must exist a point P' that has a smaller n-match difference, that is, P' has at least n dimensions smaller than dif, and then P' should have appeared n times. This result is contradictory to the fact that P is the first point that appears n times. Therefore, the supposition that P does not have the smallest n-match difference is wrong.

We can use a similar method as above to prove that the second point that appears n times must have the second smallest k-n-match difference, and so on. Therefore, the points returned by the algorithm KNMatchAD compose the k-n-match set of Q. □

Theorem 3.2. ***Optimality of KNMatchAD***
Among all algorithms that guarantee correctness for any data set instances, algorithm KNMatchAD retrieves the least attributes for the k-n-match problem.

Proof. Suppose another algorithm $\mathcal{A}$ retrieves one less attribute a than the attributes retrieved by KNMatchAD. Suppose a is dimension i of point pid_1 (for convenience, we may simply use a point ID to represent the point). $a - q_i$ must be smaller than the k-n-match difference δ (otherwise it would not be retrieved by KNMatchAD). In our model, data are sorted according to the attribute values. The algorithm only has information on the attribute value range but no information on the associated point ID at all. Therefore, as long as we keep the attribute values the same, an algorithm will retrieve the same values no matter how the associated point ID change. In other words, the set of attributes retrieved is irrespective to the point ID's. Given this observation, we can construct a data set instance as follows, which will make $\mathcal{A}$ produce wrong k-n-match answers.

Let point pid_2 be the point with the k-th smallest n-match difference, that is, it should be the last point to join the k-n-match answer set. Let b be an attribute of point pid_2 that is less than δ. Suppose b is in dimension j, hence $b - q_j < \delta$. Further, let point pid_3 be the point with the $(k+1)$-th smallest n-match difference and let this difference be smaller than point pid_2's $(n+1)$-match difference. If $\mathcal{A}$ returns the correct answer, then (b, pid_2) is already retrieved when $\mathcal{A}$ finished searching. Now consider two (attribute, point ID) pairs in the sorted dimensions: (a, pid_1) and (b, pid_2). We exchange the point ID's of these two pairs and obtain a new data set instance with (a, pid_2) and (b, pid_1), while everything else is the same as the original data set. According to our observation, $\mathcal{A}$ is not aware of the change of the point ID's, and still will not retrieve the pair with attribute a. In this case, $\mathcal{A}$ can only find $n-1$ dimensions less than δ for point pid_2. Because of not retrieving (a, pid_2), $\mathcal{A}$ thinks pid_2's $n+1$-match difference is its n-match difference, and hence will return point pid_3 as the point with the k-th smallest n-match difference. Therefore, $\mathcal{A}$ will return wrong answers if it retrieves any less attribute than KNMatchAD does. □

More generally, as long as an algorithm $\mathcal{A}$ knows nothing about the point ID before retrieving an attribute (the dimensions not necessarily sorted), $\mathcal{A}$ still have to retrieve all the attributes that KNMatchAD retrieves to guarantee correctness for any data set. The proof is the same as above. The multiple system information retrieval model satisfies the condition here, therefore KNMatchAD is optimal among all the algorithms that search k-n-match correctly, including those not based on attributes sorted at each system.

3.2 The AD Algorithm for Frequent K-N-Match Search

For frequent k-n-match search, the AD algorithm works in a similar fashion as for k-n-match search. The difference is that, instead of monitoring point ID's that appear n times, we need to monitor point ID's whose number of appearances are in the range $[n_0, n_1]$.

The AD algorithm for frequent k-n-match search, namely "FKNMatchAD", is illustrated in Figure 6. Line 1 initializes some structures used in the algorithm. *appear*[ ], $h[\ ]$ and $S[\ ]$ have the same meanings as in algorithm KNMatchAD except that h and S are arrays, each has d elements. After initialization, we locate the query's attributes in each dimen-

```
Algorithm FKNMatchAD
Initialize $appear[\,]$, $h[\,]$ and $S[\,]$.
for every dimension $i$
    Locate $q_i$ in dimension $i$.
    Calculate the differences between $q_i$ and its
    closest attributes in dimension $i$ along both
    directions. Form a triple $(pid, pd, dif)$ for each
    direction. Put this triple to $g[pd]$.
do
    $(pid, pd, dif)$ = smallest($g$);
    $appear[pid]$++;
    if $n_0 \le appear[pid] \le n_1$
        $h[appear[pid]]$++;
        $S[appear[pid]]$=$S[appear[pid]] \cup pid$;
    Read next attribute from dimension $pd$ and form
    a new triple $(pid, pd, dif)$. If end of the dimension
    is reached, let $dif$ be $\infty$. Put the triple to $g[pd]$.
while $h[n_1] < k$
scan $S_{n_0}$, ..., $S_{n_1}$ to obtain the $k$ point ID's that
appear most times
End FKNMatchAD
```

Figure 6: Algorithm FKNMatchAD

sion and put the $2d$ attributes with smallest differences to the query into the array $g[\,]$. Next we retrieve (pid, pd, dif) triples from $g[\,]$ in ascending order of dif and update $h[\,]$ and $S[\,]$ accordingly. We keep doing this until there are k points that have appeared n_1 times.

Before k points appear at least n_1 times, they must have already appeared n_0 times, ..., n_1-1 times. Therefore, when algorithm FKNMatchAD stops, that is, when it finds the k-n_1-match answer set, it must have found all the k-i-match answer sets, where $i = n_0, ..., n_1$. Then we simply need to scan the k-i-match answer sets for $i = n_0, ..., n_1$ to get the k points that appear most frequently. This shows the correctness of the algorithm. At the same time, we can see that algorithm FKNMatchAD retrieves the same number of attributes as if we are performing a k-n_1-match search by algorithm KNMatchAD. Since we have to at least retrieve the attributes necessary for answering the k-n_1-match query, and we only need to retrieve this many to answer the frequent k-n-match query, we consequently have the following theorem:

THEOREM 3.3. ***Optimality of FKNMatchAD***
Algorithm FKNMatchAD retrieves the least attributes for the frequent k-n-match problem.

We can see that the frequent k-n-match search is no harder than a k-n-match search with the same k and n values. However, the frequent k-n-match query can take advantage of the results of a range of n values to obtain answers based on full similarity.

4. DISK BASED SOLUTIONS

In the previous section, we have investigated AD algorithms that are optimal in the model where cost is measured by the number of attributes retrieved. This model directly applies to the multiple system information retrieval problem. In this section, we would like to study how the AD algorithm works in the disk based model. Our cost measure still conforms with the disk model, where the number of disk accesses is the major measure of performance, and disk accesses is proportional to the attributes retrieved. However, one complication is that disk based algorithms may make use of auxiliary structures such as indexes or compression to prune data. R-tree based approaches have been shown to perform badly with high dimensional data due to too much overlap between page regions, and also no other indexing techniques can be applied directly to out problem because of the *dynamic* dimensions used for aggregating the score. Only compression techniques still apply such as the one used in VA-file [21], which does not rely on a fixed set of dimensions. Therefore, we will describe the disk based AD algorithm and an adaption from the VA-file algorithm for our problem in the following. We will use the VA-file based algorithm as a competitor when evaluating the efficiency of the AD algorithm in the experimental study.

4.1 Disk Based AD Algorithm

As we can see from Section 3.2, algorithm KNMatchAD is actually a special case of FKNMatchAD (when $n_0 = n_1$). Therefore we will focus on the frequent k-n-match search. First, we sort each dimension and store them sequentially on disk. Then we can use the same FKNMatchAD algorithm except that, when reading the next attribute from the sorted dimensions, if we reach the end of a page, we will read the next page from disk. Note that FKNMatchAD accesses the pages sequentially when search forwards, which makes the processing more efficient.

4.2 Compression Based Approach

Compression based techniques such as the VA-file [21] can be adapted to process the frequent k-n-match query. The algorithm runs in two phases. The first phase scans an approximation of the points, that is, the VA-file. During the first phase, the algorithm calculate lower and upper bounds of the k-n-match difference of each point based on the VA-file and utilizes these bounds to prune some points. Only the points that go through the first phase will be actually retrieved from the database for further checking in the second phase. We omit the implementation details here.

5. EXPERIMENTAL STUDY

In this section, we evaluate both the effectiveness and the efficiency of the (frequent) k-n-match query by an extensive experimental study. We use both synthetic uniform data sets and real data sets of different dimensionalities. The data values are all normalized to the range [0,1]. All the experiments were run on a desktop computer with 1.1GHz CPU and 500M RAM.

5.1 Effectiveness

To validate the statement that the traditional kNN query leaves many partial similarities uncovered, we first use a image database to visually show this fact in Section 5.1.1. In comparison, we show that the k-n-match query can identify similar objects by partial similarities if a proper n is chosen. But note that we are not using this to argue the effectiveness of the k-n-match approach for full similarity. The technique we use for full similarity search is the frequent k-n-match query and we will evaluate its effectiveness statistically in Section 5.1.2.

5.1.1 Searching by K-N-Match

We shall first visually show that the k-n-match query yields better result than the k-n-match search if a proper value of n is chosen. To do so, we use the COIL-100 database [1], which consists of 100 images. Some of the images in the database are shown in Figure 7 (the numbers under the images are the image ID's). We extracted 54 features from these images such as color histograms and moments of area. Below we show a sample of the experiments we conducted and the results of other searches on the database exhibit similar behavior.

In this experiment, we used image 42 as the query object.

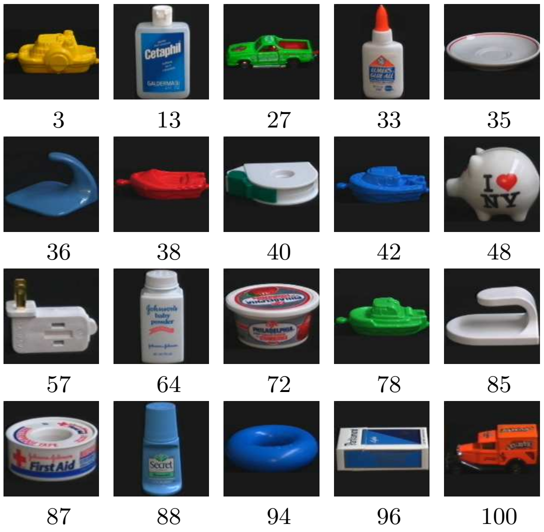


Figure 7: Images in the COIL-100 database

Table 2: k-n-match results, $k = 4$, Query Image 42

n	images returned		n	images returned
5	36, 42, 78, 94		30	10, 35, 42, 94
10	27, 35, 42, 78		35	35, 42, 94, 96
15	3, 38, 42, 78		40	35, 42, 94, 96
20	27, 38, 42, 78		45	35, 42, 94, 96
25	35, 40, 42, 94		50	35, 42, 94, 96

Table 3: kNN results, $k = 10$, Query Image 42

k	images returned
10	13, 35, 36, 40, 42 64, 85, 88, 94, 96

Table 2 shows the results returned by k-n-match with $k = 4$ and sampled n values varying from 5 to 50. The results of the kNN search is shown in Table 3 and the 10 nearest neighbors returned based on Euclidean distance are given. Comparing the two tables, the most obvious difference is the existence of image 78 in the k-n-match frequently which is not found in the 10 nearest neighbors of kNN search. Image 78 is a boat which is obviously more similar to image 42 compared to images 13, 64, 85, and 88 in the kNN result set. In fact, we did not find image 78 in the kNN result set even when finding 20 nearest neighbors. The difference in color between image 42 and 78 is clearly dominating all other aspects of comparison. The k-n-match query successfully identifies this object because of the use of partial matches.

Among the remaining k-n-match result, perhaps less noticeable is the existence of image 3. It is obviously more similar to image 42 than many images in the kNN result set and image 3 is in fact a yellow color and bigger version of image 42. However, it appears only once in the k-n-matches of different n values. If we are not using a good n value, we may miss this answer.

As can be seen from these results, k-n-match can yield better result than kNN search, but it also depends on a good choice of n. This motivates the use of the frequent k-n-match query, which returns objects that have many partial matches with the query object.

5.1.2 Searching by Frequent K-N-Match

We next evaluate the effectiveness of our proposed method, the frequent k-n-match query, for finding objects of full similarity. In order to evaluate effectiveness from a (statistically) quantitative view, we use the *class stripping technique* [6], which is described as follows. We use five real data sets from the UCI machine learning repository [2] with dimensionalities varying from 4 to 34: 1) the ionosphere data set contains 351 34-dimensional points with 2 classes; 2) the image segmentation data set contains 300 19-dimensional points with 7 classes; 3)the wdbc data set contains 569 30-dimensional points with 2 classes; 4) the glass data set contains 214 9-dimensional points with 7 classes; 5) the iris data set contains 150 4-dimensional points with 3 classes. Each record has an additional variable indicating which class it belongs to. By the class stripping technique, we strip this class tag from each point and use different techniques to find the similar objects to the query objects. If the answer and the query belong to the same class, then the answer is correct. The more correct ones in the returned answers, statistically, the better the quality of the similarity searching method.

We run 100 queries which are sampled randomly from the data sets, k set as 20. We count the number of the answers with correct classification and divide it by 2000 to obtain the accuracy rates. Two techniques proposed previously: IGrid [6] and the Human-Computer Interactive NN search (HCINN for short) [4] have been shown to obtain more accurate results than the traditional kNN query. Therefore, we will compare the frequent k-n-match query with these two techniques. $[n_0, n_1]$ for the frequent k-n-match query is simply set to $[1, d]$. The results are shown in Table 4. As the code of HCINN is not available, its accuracies on the ionosphere and segmentation data sets are adopted directly from [4] while results on other data sets are not available.

We can see that frequent k-n-match constantly obtains higher accuracy than the other two techniques. It improves the accuracy up to 9.2% over IGrid. Therefore, we argue that the frequent k-n-match query is a more accurate model for similarity search.

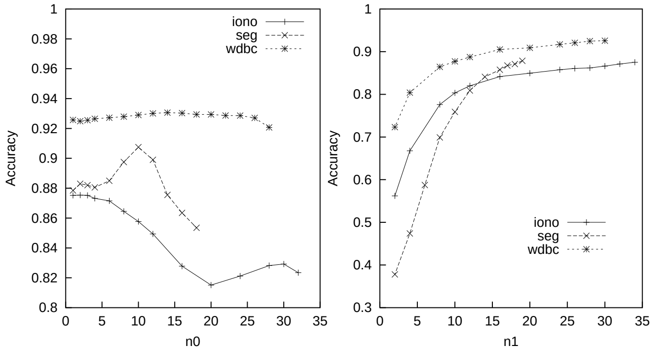
Table 4: Accuracy on Real data sets

data sets (d)	IGrid	HCINN	Freq. k-n-match
Ionosphere (34)	80.1%	86%	87.5%
Segmentation (19)	79.9%	83%	87.3%
Wdbc (30)	87.1%	N.A.	92.5%
Glass (9)	58.6%	N.A.	67.8%
Iris (4)	88.9%	N.A.	89.6%

5.2 Efficiency

In Section 3, we have proved that the AD algorithm is optimal in terms of number of attributes retrieved. However, is it also efficient in a disk based model? To answer this question, we would like to conduct the following experimental studies. First, we will study how to choose parameters, particularly, the range of frequent k-n-match, $[n_0,n_1]$, to optimize its performance (we will focus on frequent k-n-match instead of k-n-match, since frequent k-n-match is the technique we finally use to perform similarity search). Second, we will study, using well chosen parameters, which searching scheme is the best for frequent k-n-match search. Third, we would like to study how the efficiency of frequent k-n-match search is compared to other similarity search techniques, such as IGrid, since IGrid is more effective than kNN and can be processed very efficiently as reported in [6].

5.2.1 Choosing Parameters

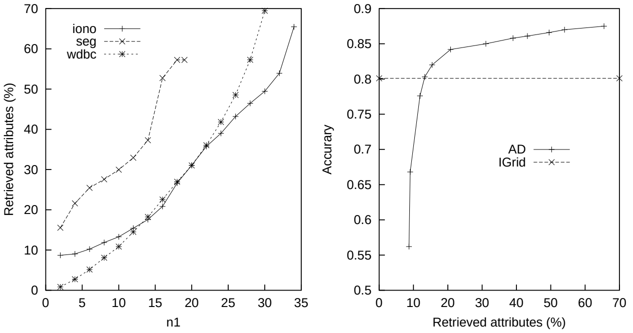


(a) Accuracy vs. n_0 (b) Accuracy vs. n_1

Figure 8: Effects of n_0 and n_1

Figure 8 illustrates the effects of the range of the frequent k-n-match, $[n_0,\ n_1]$, on the accuracy of the results of the three high dimensional machine learning data sets: ionosphere (iono), image segmentation (seg), and wdbc, still using the class stripping technique described in Section 5.1.2. Figure 8 (a) plots the accuracy as a function of n_0 while fixing n_1 as d. We can see that, as n_0 increases, the accuracy first increases, and then decreases. This is because when n is too small, there are not enough attributes to capture any feature of the object but some random matches. Using such small n values decreases the accuracy. When there are enough number of dimensions, the results begin to make sense and accuracy increases. However, when n_0 is too large, the range of $[n_0,\ n_1]$ becomes too small to identify frequently appearing objects, and therefore the accuracy decreases again. As the accuracy on the ionosphere data set starts to decrease from $n_0 = 4$, we have chosen n_0 conservatively as 4 in the following experiments.

Figure 8 (b) shows the accuracy of frequent k-n-match as a function of n_1 while fixing the value of n_0 as 4. The accuracy decreases as n_1 decreases. This is expected since the larger the range, the more stable the frequent appearing objects that we find. We observe that the accuracy decreases very slowly when n_0 is large. As n_0 becomes smaller, the accuracy decreases more and more rapidly. The reason is that, when n is large, more dimensions of high dissimilarities are taken into account. These dimensions do not help in finding similarities between objects.



(a) Attr retrieved vs. n_1 (b) Accuracy vs. Attr retrieved

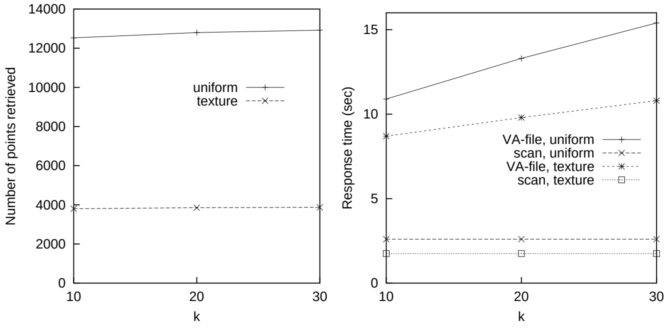
Figure 9: Tradeoff between accuracy and performance

In another experiment, we would like to see the relationship between the number of attributes retrieved and n_1, which is revealed in Figure 9. This figure shows that there is a tradeoff between the accuracy and performance of the AD algorithm in terms of number of attributes retrieved. Figure 9 (a) plots the number of attributes retrieved (in terms of percentage of the cardinality of the data set) by the AD algorithm as a function of n_1. The number of attributes retrieved increases as n_1 increases since the larger the n_1, the larger the k-n-match difference and hence the more attributes smaller than this k-n-match difference. An interesting phenomenon is that, in contrary to the trend of the accuracy, the increase of the number of attributes retrieved is slower when n_1 is small than when n_1 is large. This means that, by decreasing n_1 slightly from d, we can achieve a large performance gain without sacrificing much accuracy. And we have plotted this tradeoff between accuracy and performance more clearly in Figure 9 (b). This figure shows the accuracy of the AD algorithm versus the percentage of attributes retrieved on the ionosphere data set. We can see that the accuracy increases most rapidly when about 10% of the attributes are retrieved. After this, the accuracy increases much slower. We also draw the accuracy of the IGrid technique on the same data set. When the AD algorithm achieves the same accuracy as IGrid, less than 15% of the attributes are retrieved. Results on other data sets have the similar trend and all retrieve about 15% attributes when getting the same accuracy as IGrid. Therefore, we choose the n_1 value according to the accuracy of IGrid when comparing efficiency with IGrid. By this means, n_1 is about 8 for the high dimensional real data sets, varying 1 or 2 depending on the dimensionality.

5.2.2 Evaluation of Disk Based Algorithms for Frequent K-N-Match

As the data sets used for the above studies are too small for run time testing on disk based solutions (the queries finish too fast to make any difference in time for different techniques), we use data sets with more points for efficiency evaluation. We generated uniformly distributed data sets of various dimensionalities and also a real data set, the Cooccurrence Texture from the UCI KDD archive[3]. All uniform data sets contain 100,000 points. The Texture data set contains 68040 16-dimensional points. Data page size is 4096 bytes. Because frequent k-n-match search is the final technique we use to performance similarity search, we focus on frequent k-n-match search instead of k-n-match search. The range $[n_0, n_1]$ of frequent k-n-match search is chosen according to the results on real data sets as described in Section 5.2.1.

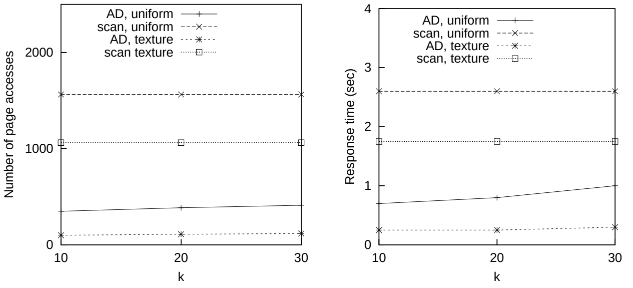
First, we evaluate the VA-file based algorithm as described in Section 4.2. In our implementation of the VA-file, we use 8 bits to code the data, which make the size of VA-file 25% of the size of the original data set. Figure 10 shows the results on a 16-dimensional uniform and the Texture data sets. Figure 10 (a) shows the number of points that are actually retrieved from the database in the refinement phase of the VA-file based algorithm for frequent k-n-match. As the total number of points is 100,000 and 68,040 for the uniform and the Texture data sets respectively, there are about 10% of points retrieved. For these about 10% points, the algorithm needs to do random page accesses to retrieve them, therefore the final response time turns out to be about twice that of the scan algorithm, as shown in 10 (b). Results of data sets of other dimensionalities have similar behavior. Therefore, VA-file based algorithm does not work for the frequent k-n-match query.



(a) Number of points retrieved (b) Response time

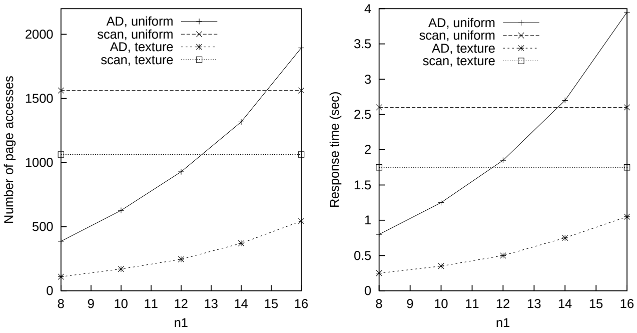
Figure 10: Performance of VA-file based algorithm

Next, we evaluate our proposed AD algorithm. The number of page accesses and response time on a 16-dimensional uniform and the Texture data sets are shown in Figure 11 (a) and (b), respectively. The number of page accesses of AD is 10∼20% of the sequential scan and the result of response time is similar. Because the AD algorithm retrieves only the necessary attributes for evaluating the frequent k-n-match query and search forwards in a dimension take advantage of sequential accesses, it beats sequential scan on the total response time. This shows the efficiency of the AD algorithm.



(a) Page access (b) Response time

Figure 11: Performance of the AD algorithm



(a) Page access vs. n_1 (b) Response time vs. n_1

Figure 12: Performance of the AD algorithm

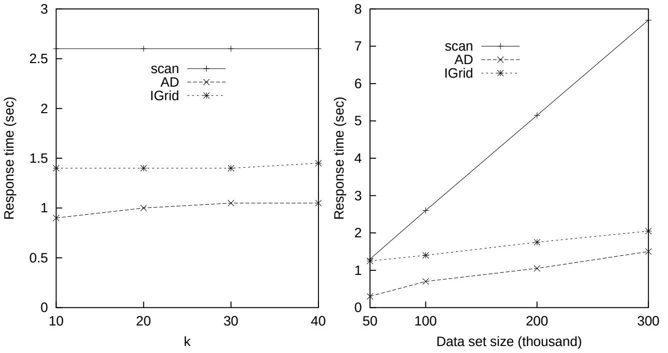
We also plotted the number of page accesses and response time as functions of n_1 in Figure 12 (a) and (b), respectively. While the AD algorithm can achieve the same accuracy as IGrid when n_1 as low as 8, the AD algorithm beats the sequential scan even when n_1 is much larger (up to 14). This means that our technique can achieve high accuracy in similarity search while still being very efficient.

From the above comparison with VA-file based algorithm and sequential scan, we can draw the conclusion that the AD algorithm is still the best choice among the competitors in the disk based model.

5.2.3 Comparison with Other Similarity Search Techniques

In this section, we compare the efficiency of the frequent k-n-match query using the AD algorithm (that is, FKNMatchAD) with other similarity search techniques. Both IGrid [6] and the Human-Computer Interactive NN search (HCINN for short) [4] have been reported to have better accuracy than the kNN query. We have shown that frequent k-n-match search has better accuracy than them in Section 5.1.2. Therefore, we would like to further see how is the efficiency of our method compared with these two techniques.

The HCINN search algorithm needs to access all the data in the data set and moreover, it requires human interaction, therefore it is less efficient than FKNMatchAD. In the following, we will only compare FKNMatchAD with IGrid. IGrid [6] was proposed as an inverted file on the grid partition of the database. The analysis in [6] shows that the accessed data size is 2/d of the original data, therefore the data accessed decreases as the dimensionality increases. However, in their analysis, they only considered the sum of the size of the data accessed, but not how the data are distributed on the disk. In fact, the accessed data are fragmented and distributed all over the data set. Random accesses of all the fragments are much more expensive than when they are clustered together and accessed sequentially. So a mere comparison in the size of the accessed data is not enough to show its efficiency. In view of this, we have compared the response time of FKNMatchAD and IGrid, using both synthetic and real data sets.



(a) Response time vs. k (b) Response time vs. dataset size

Figure 13: Comparison with IGrid

The response time of the two techniques, FKNMatchAD and IGrid, on a 16-dimensional uniform data set with varying k and data set sizes are shown in Figure 13. We also plotted the response time of the sequential scan algorithm for frequent k-n-match search as a reference for FKNMatchAD. We see that the FKNMatchAD is more efficient than IGrid. And FKNMatchAD is scalable with regard to k and data set size. We also compared them for on data sets of varying dimensionalities from 8 to 48. FKNMatchAD always outperforms the other two techniques as shown in Figure 14.

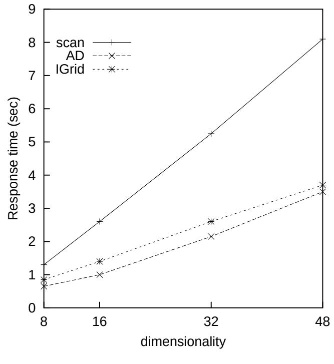
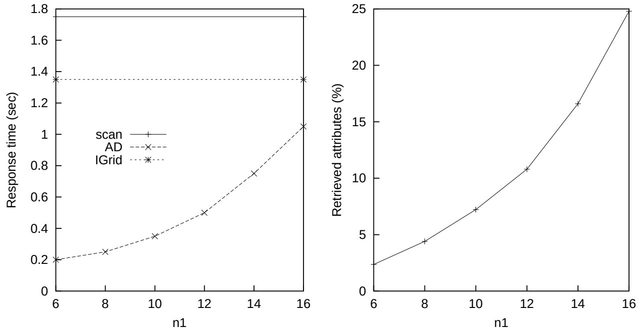


Figure 14: Effect of dimensionality

Finally, we compare them on the real data set (the Texture data set). The result of response time is shown in Figure 15 (a). We can see that FKNMatchAD beats the other two techniques even when n_1 equals the dimensionality 16. By examining the number of attributes retrieved as shown in Figure 15 (b), we can see that when $n_1 = 16$, there is only 25% of the attributes retrieved due to the high skew of the real data. This is the reason for the especially good performance exhibited here.



(a) Response time vs. n_1 (b) Attributes retrieved vs. n_1

Figure 15: Comparison with IGrid on real data

From the above results, we draw the conclusion that the frequent k-n-match query can be processed more efficiently (by our proposed FKNMatchAD algorithm) than the existing techniques while achieving better accuracy than them in similarity search.

6. RELATED WORK

A popular method for similarity search is to first extract from objects some features such as image colors [14], shapes [17] and texts [19], and then use nearest neighbor queries to search similar objects [10, 14]. In the last decade, many structures and algorithms have been proposed aiming at accelerating the processing of (k) nearest neighbor queries. Early methods are based on R-tree-like structures such as the SS-tree [22] and the X-tree [7]. However, the R-tree-like structures all suffer from the "dimensionality curse", that is, their performance deteriorates dramatically as dimensionality becomes high. [21] has shown this phenomenon both analytically and experimentally. Therefore, the authors of [21] proposed an algorithm based on compression, called the vector approximation-file (VA-file) to accelerate sequential scan.

While the papers above mainly emphasize on the efficiency of kNN search, other works look at kNN from the aspect of effectiveness. In [8], Beyer et. al. show that at very high dimensionality, the distance between two nearest points and two furthest points in a data set are almost the same. At the same time however, they also show that points that are generated from distinct clusters do not obey such rules. Various studies [16, 5, 6, 4] have been performed subsequently to address the issue raised in [8]. Among these, only [6] addresses the efficiency issue. In [6], the IGrid index was proposed, in which each dimension is discretized based on equi-depth partitioning in a pre-processing phase. When comparing two points, the actual difference between matching dimensions are aggregated to judge their similarity. This is different from our work which performs the discretization dynamically while counting only the matches. The most effective among these works is reported in [4] where human computer interaction is needed to find meaningful neighbors.

In [18], dynamic partial function (DPF) was proposed to compute similarity based on the closest n dimensions. Our work employs the similar strategy in defining the n-match problem. However, in view of the hardness to define a good n value in reality, we propose the frequent k-n-match problem which captures the full similarity of objects and the result is not sensitive to the choice of n. [18] used an n value observed from experiments over the data set, which is an ad hoc method. Moreover, the algorithm proposed in [18] has no correctness guarantee and their accuracy is measured by recall of the actual kNN, that is, how many actual kNNs are included in their answers. In other words, the algorithm finds approximations to the exact kNN answers, but without any approximation guarantee. This is different from our effectiveness evaluation, which measures the extent of similarity of answers to the query, while the answers are exact correct nearest neighbors under our similarity model.

We have discussed our problem in the multiple system information retrieval model, which was described in [11]. As discussed in Section 3, the algorithm proposed in [11] and variations in [13] were for other types of queries and they assume a monotone aggregation function, which is not satisfied by the aggregation function of our problem. More recently, [12] has used rank aggregation to answer kNN approximately and the quality measure is the approximation factor. Again, the difference is that we are answering the query defined by our similarity model exactly.

The skyline query [9, 20] has also been proposed to find close objects based on various feature sets, but the answer of the skyline query is a set of objects that do not dominate each other. In our model, we still find top k answers according to one score, the n-match difference. In this sense, it is closer to the traditional kNN query that the answer with higher score dominates the ones with lower scores.

7. CONCLUSION

In this paper, we proposed a new approach to model the similarity search problem, namely the k-n-match problem. The k-n-match problem models the similarity search as matching between the query object and the data objects in n dimensions, where these n dimensions are determined dynamically to make the query object and the data objects in the answer set match best. While the k-n-match query is expected to be superior than the kNN query in discovering partial similarities, it depends on a good choice of the n value. To address the problem, we further introduced the *frequent k-n-match* query, which returns the objects that appear most frequently in the answer sets of k-n-match queries with a range of n values. Moreover, we proposed algorithms (called the AD algorithm) for both problems. We proved that the AD algorithm is optimal in the multiple system information retrieval model. We also applied the strategy to obtain a disk based algorithm for the (frequent) k-n-match query. By a thorough experimental study using both real and synthetic data sets, we validated that the k-n-match query finds better result than the kNN query through partial similarity if a good value of n is chosen; we showed that the frequent k-n-match query is more effective in similarity search than existing techniques such as IGrid and Human-Computer Interactive NN search, which have been reported to be more effective than traditional kNN queries based on Euclidean distance. We also showed that the frequent k-n-match query can be processed more efficiently than the other techniques by our proposed AD algorithm in the disk based cost model.

8. REFERENCES

[1] *http://www1.cs.columbia.edu/CAVE/research/softlib/coil-100.html.*

[2] *ftp://ftp.ics.uci.edu/pub/machine-learning-databases/.*

[3] *http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures.data.html.*

[4] C. C. Aggarwal. Towards meaningful high-dimensional nearest neighbor search by human-computer interaction. In *ICDE*, 2002.

[5] C. C. Aggarwal, A. Hinneburg, and D. A. Keim. On the surprising behavior of distance metrics in high dimensional spaces. In *ICDT*, 2001.

[6] C. C. Aggarwal and Philip S. Yu. The igrid index: Reversing the dimensionality curse for similarity indexing in high dimensional space. In *KDD*, 2000.

[7] S. Berchtold, D. Keim, and H.-P. Kriegel. The x-tree: An index structure for high-dimensional data. In *VLDB*, 1996.

[8] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is nearest neighbors meaningful? In *ICDT*, 1999.

[9] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, 2001.

[10] T. Chiueh. Content-based image indexing. In *VLDB*, 1994.

[11] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, 1996.

[12] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *SIGMOD*, 2003.

[13] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[14] C. Faloutsos, W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and R. Barber. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3(3):231–262, 1994.

[15] Richard W. Hamming. Error detecting and error correcting codes. *Bell Systems Technical Journal*, 29:147–160, 1950.

[16] A. Hinneburg, C. C. Aggarwal, and D. A. Keim. What is the nearest neighbor in high dimensional spaces? In *VLDB*, 2000.

[17] H. V. Jagadish. A retrieval technique for similar shapes. In *SIGMOD*, 1991.

[18] E. Y. Chang K.-S. Goh, B. Li. Dyndex: a dynamic and non-metric space indexer. In *ACM Multimedia*, 2002.

[19] Karen Kukich. Techniques for automatically correcting words in text. *ACM Computing survey*, 24(4):377–439, 1992.

[20] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *TODS*, 30(1):41–82, 2005.

[21] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, 1998.

[22] D. A. White and R. Jain. Similarity indexing with the ss-tree. In *ICDE*, 1996.