Similarity Search: A Matching Based Approach

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

[†] National Univ. of Singapore [‡] Univ. of Melbourne [§] Univ. of Toronto
{atung, ooibc}@comp.nus.edu.sg rui@csse.unimelb.edu.au 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.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '06, September 12-15, 2006, Seoul, Korea.
Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09.

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

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

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

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.

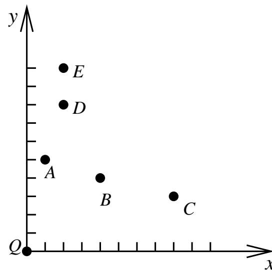

Figure 2: The n-match problem

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

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

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

$i = 1, ..., d$). 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) = smallest(g);
7      appear[pid]++;
8      if appear[pid] = n
9          h++;
10         S=S ∪ 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 ∞. 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.