A Locality-Aware Similar Information Searching Scheme

Ting Li
Dept. of Computer Science and Computer Engineering
University of Arkansas, Fayetteville, AR 72701
txl005@uark.edu

Haiying Shen
Dept. of Electrical and Computer Engineering
Clemson University, Clemson, SC 29631
shenh@clemson.edu

Abstract

In a database, a similar information search means finding data records which contain the majority of search keywords. Due to the rapid accumulation of information nowadays, the size of databases has increased dramatically. An efficient information searching scheme can speed up information searching and retrieve all relevant records. This paper proposes a Hilbert Curve based similarity Searching scheme (HCS). HCS considers a database to be a multidimensional space and each data record to be a point in the multidimensional space. By using a Hilbert space filling curve, each point is projected from a high dimensional space to a low dimensional space so that the points close to each other in the high dimensional space are gathered together in the low dimensional space. Because the database is divided into many clusters of close points, a query is mapped to a certain cluster instead of searching the entire database. Experimental results prove that HCS dramatically reduces the search time latency and exhibits high effectiveness in retrieving similar information.

Keywords: Hilbert curve, Locality sensitive hashing, Similarity Searching, Massive databases

1 Introduction

A database aims to store data objects and provide access to the content of the data objects. These objects are called records, and they are represented in the database by various attributes associated with the objects as independent dimensions. Therefore, the data objects are mapped into a high-dimensional data space. For example, a text document may be represented by the word frequencies of a very large vocabulary; images may be described by the features such as shape, colour and texture. As the cardinality of data sets increases, efficient high-dimensional data querying becomes increasingly important. One example of querying high-dimensional data is similarity search. In essence, similarity search is retrieving the objects which are similar to the query object for a given degree. For example, two records A and B:

A: Ann Johnson 16 Female

B: Ann Smith 20 Female

Because both A and B contain the keywords "Ann" and "Female", A and B are similar records, as they contain similar information. A few applications of similarity search include audio and image databases [15], video, text files, fingerprints [30], face recognition [29], and protein sequences [8]. In many cases, the high dimensional space is Euclidean space [25]. Given a query object q, a database S of objects s_i, the number of objects n, and a metric distance function $d(x, y)$ (i.e., Euclidean distance computation function), the objects that satisfy any of the following conditions can be located as the similar objects of a query object q.

1. The object is closest to the query object q, i.e., $\{s_j \in S | \forall s_i \in S : d(s_j, q) \leq d(s_i, q)\}$ (nearest neighbour query).
2. The first k objects are closest to the query object q, where $0 < k < n$ (k-nearest neighbour query).
3. The object whose distance with query object q falls within a given range r, i.e., $\{s_j \in S | d(s_j, q) \leq r\}$ (range query).

It is well known that effective indexing of the data is necessary for efficient query processing. Because of the curse of dimensionality [26], indexing high dimensional

data is a harder problem. Often as the dimensionality of the space increases, the difference in the distance between the nearest and the farthest objects decreases [5]. Searching in a high-dimensional space can be time-consuming. For similarity search, $O(n)$ (n is the number of objects in the database) time is needed to compute the distance between the query object and every object in the database, which is not viable for a large database. This has motivated the development of efficient similarity search techniques, such as kd-tree [4], R-tree [19], VP-tree [16] and Bk-tree [6].

To speed-up the search, a trade-off between searching quality and searching latency is offered [32]. An acceptable degradation in the quality of the searching results can save searching time. Santini and Jain [35] gave an example of similarity queries over multimedia data with consideration of the tradeoff. Therefore, a similarity search result may contain objects that are not similar to the query object, called false positives. Similarly, an object that is similar to the query object is named as true positive.

This paper proposes a Hilbert Curve based similarity Searching scheme (HCS) which can cluster records according to their similarity. A Hilbert curve [22] is a space-filling curve [33]. It is used in image processing, especially image compression and dithering. A Hilbert curve is employed in HCS because of its local order preservation property. It can project high dimensional data points into a low dimensional space. HCS assigns a Hilbert number for each record, and then uses the Hilbert curve's locality preserving property to cluster similar records. Queries are conducted in the clusters that have the same Hilbert numbers as the queries. We conduct experiments to investigate the operation of HCS and compare the performance of HCS with linear search. The dimension of the testing data set is 33,601. Experiment results show that HCS is an efficient similarity searching scheme. Compared with linear search, HCS dramatically reduces the query time.

The rest of this paper is structured as follows. Section 2 presents a concise review of similarity searching methods. Section 3 presents the design of HCS. Section 4 shows the experimental results. Finally, Section 5 draws conclusions and summarizes the propositions of the HCS scheme.

2 Related Work

As the dimensionality of the space increases, the difference in the distance between the nearest and the farthest objects decreases [5]. The "curse of dimensionality" [26] makes it hard to index high dimensional data. In order to tackle the "curse of dimensionality", various approximate solutions based on dimensionality reduction have been proposed [2] [14] [17].

Locality sensitive hashing (LSH) [3] [10] [11] [17] [21] [28] is a method for performing nearest neighbour searches. LSH is developed in [17]. Its key idea is to hash the points using a family of hash functions so that the probability of close points being hashed into the same value is much higher than that of distant points. With LSH, close neighbours of a query point can be determined by retrieving elements with similar hashed values to the query point's hashed value. For filtering the search results, the Euclidean distance is computed between the query point and every retrieved point. The points whose distances are greater than a predefined threshold are removed from the results. However, one of the main drawbacks of LSH is the large memory requirement. LSH must consume large memory resources to achieve fast query speeds. The second drawback of LSH is that Euclidean distance computation leads to long query times and the distance computation phase is indispensable for LSH in a high dimensional space.

Another nearest neighbour searching method relies on tree structures. R-trees (Rectangle trees) were proposed as an extension of B-trees; they are used as dynamic index structures for spatial searching [19]. An R-tree uses an n-dimensional rectangle that is the bounding box to bind each data object. Each node of an R-tree has many entries. Each entry within a non-leaf node stores the address of a child node and a minimum bounding rectangle (MBR) of all entries within this child node. Leaf nodes contain pointers to the data objects and their enclosing rectangles [27]. The SS-tree (Similarity Search tree) is similar to an R-tree. Instead of using MBR, SS-tree [37] employs minimum bounding spheres (MBS), which can reduce the requirement for storage. The objects are grouped together by spheres in a hierarchical manner. The parent node's sphere completely bounds all the spheres of the nodes beneath it in the tree [12]. The SR-tree (Square/Rectangle tree) [24] [12] utilizes both MBSs and MBRs to represent

the minimum bounding region, which is the intersection of MBRs and MBSs. A leaf node of the SR-tree contains many entries, and each entry contains a point and its attribute data. A non-leaf node also consists of a number of entries. Each entry corresponds to a child node and consists of four components: a bounding sphere, a bounding rectangle, the number of points, and a pointer to the child node. This improves search efficiency over R-trees and SS-trees. However, as reported in [5], the performance of an SR-tree is not as good as a sequential scan when dimensionality is greater than 20.

The M-tree [9] was proposed to organize and search large data sets from a generic metric space, i.e., where object proximity is only defined by a distance function satisfying the positivity, symmetry, and triangle inequality postulates. The M-tree partitions objects on the basis of their relative distances as measured by a specific distance function and stores these objects into fixed-size nodes that correspond to constrained regions of the metric space [9]. All data objects are stored in the leaf nodes of an M-tree. The non-leaf nodes contain "routing objects," which describe the objects contained in the branches. For each routing object, there is a so-called covering radius for all of its enclosing objects, and the distances to each child node are pre-computed. When a range query is completed, sub-trees are pruned if the distance between the query object and the routing object is larger than the routing object's covering radius plus the query radius. Because a lot of the distances are pre-computed, the query speed is dramatically increased. The main problem is the overlap between different routing objects in the same level [31].

The vector approximation file (VA-file) [36] can reduce the amount of data that must be read during a similarity search. VA-file does not use a tree structure but instead stores an approximation of the vector of each data object in a sequential file [12]. It divides the data space into grids and creates an approximation for each data object that falls into a grid. When searching for near neighbours, VA-file sequentially scans the file containing these approximations, which is smaller than the size of the original data file. This allows most of VA-file's disk accesses to be sequential, which is much less costly than random disk accesses [13]. One drawback of this approach is that VA-file requires a refinement step, where the original data file is accessed using random disk accesses [13].

3 Hilbert Curve based Searching Scheme

The challenge of nearest neighbour search is to effectively group similar information into the same cluster. We propose a Hilbert curve based nearest neighbour searching scheme (HCS). In particular, we assign a Hilbert number to each record in the database. Because a Hilbert curve has a locality preserving feature, the Hilbert numbers of similar records are close to each other. We group records with close Hilbert numbers into a cluster. For a query record, HCS searches the cluster of the query's Hilbert number and locates the records that have similar Hilbert numbers.

In the following sections, we first introduce space-filling curves. We then describe how to represent a record in n-dimensional space and how to use the Hilbert curve to map records from a high dimensional space to one dimensional space and cluster the similar records using a hash table. Finally, we present the similar information searching process of HCS.

3.1 Introduction to Space-Filling Curves

Space-filling curves have garnered increasing interest in recent years due to their uses in practical applications [20] [33]. Mokbel and Aref describe a space-filling curve as a "thread that goes through all the points in a space but visiting every point only once" [7]. Using this mapping, a point in n-dimensional space can be described by its spatial coordinates, or by the length along the thread, measured from one of its ends.

There are many space-filling curves available, including the Peano, Z, Hilbert, sweep, scan, and gray curves [7]. The Hilbert space-filling curve is believed to achieve the best clustering [1] [23]. A Hilbert curve partitions the n-dimensional space into 2^{nx} grids. n represents the dimensionality of the space and x controls the number of grids used to partition the multidimensional space. Figure 1 shows an example of transforming 3-dimensional points into a Hilbert space-filling curve. The points that are close to each other in 3-dimensional space are still close to each other after being projected onto a Hilbert curve.

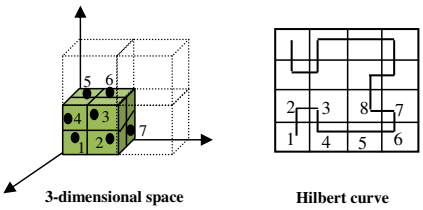


Figure 1: An example of a space-filling curve.

3.2 Multidimensional Keyword Space Construction

A Hilbert curve can transform n-dimensional spatial coordinates of points into one-dimensional indices while preserving the locality relationship between points. Therefore, to apply the Hilbert curve to the data objects in a database, all data objects need to be represented by coordinates in the same multidimensional space. However, a data object in a database is represented by a string consisting of a number of attributes, and the number of attributes in a data object differs for different objects. For example, a data object is expressed as "ANN 16 FEMALE 22 MAIN STREET". This poses a challenge to represent every object by n-dimensional spatial coordinates (a vector), i.e., to represent each object as a point in a unified n-dimensional space. The challenge is more formidable if the data is in the form of documents in a database. To cope with this challenge, HCS constructs a multidimensional keyword space which facilitates representing each data object by a certain number of coordinates.

Information retrieval (IR) deals with text processing. The vector space model (VSM) [34] is one such information retrieval strategy. To retrieve documents relevant to a query, VSM computes a measure of similarity by defining a vector that represents each document, and a vector that represents the query. VSM uses occurrences of keywords from the keyword list in the document collection to determine the vector of the document. Consider a document collection with only two keywords, α and β. Then, there are only two components in the vectors. The first component represents occurrences of α and the second represents occurrences of β. If a document D contains one occurrence of word α and zero occurrences of word β, its vector is expressed by $< \alpha : 1, \beta : 0 >$ binary representation [18]. Therefore, the vector presentation method provided by VSM changes a string document into an attribute vector (i.e., record).

Because the records in a database are described by many keywords, we use the VSM method to transform each record into a point in a high-dimensional space. We collect all the unique keywords of all the records in the database to make a token list with each keyword representing a coordinate. The total number of unique keywords is the number of dimensions in the high-dimensional space. For instance, Figure 2 shows a point in a 3-dimensional space. Point A in the figure represents a record in the 3-dimensional keyword space. It means that the number of unique keywords in the database is 3. The vector of point A is (d_1, d_2, d_3), where d_1, d_2 and d_3 are the number of occurrences of Keyword 1, Keyword 2 and Keyword 3, respectively. Therefore, the presentation of a point in n-dimensional space is $(d_1, d_2, ..., d_n)$. For each keyword, if it appears in a record equal or more than once, "1" is marked at the corresponding component in the vector; otherwise, "0" is marked. Given a database consisting of name and address keywords as shown in Figure 3, these records are transferred into a multidimensional keyword space as shown in Figure 4.

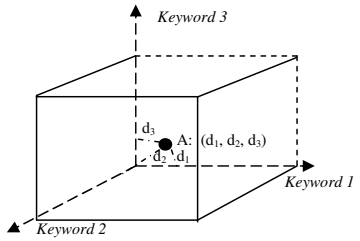


Figure 2: A point in a 3-dimensional space.

Record ID	Record
1	TOM SMITH 17 N ELM ST
2	DAVID RUFF 22 MAIN ST

Figure 3: Database of names and addresses.

Keywords \ ID	1	2
DAVID	0	1
ELM	1	0
MAIN	0	1
N	1	0
RUFF	0	1
SMITH	1	0
ST	1	1
TOM	1	0
17	1	0
22	0	1

Figure 4: Multidimensional keyword space.

Consequently, each record in the database is presented as a 10-bit series of binary numbers, where 10 is the total number of unique keywords. The vector of record 1 is $< 0, 1, 0, 1, 0, 1, 1, 1, 1, 0 >$; the vector of record 2 is $< 1, 0, 1, 0, 1, 0, 1, 0, 0, 1 >$.

3.3 Hilbert Indexing

We use a Hilbert curve to map each vector to a real number, such that the closeness relationship among the points is preserved. The following hash function is used to map a point from n-dimensional space into a Hilbert number:

$$h = H(v), \tag{1}$$

where v is the vector of a record in the database and h is the Hilbert number. For example, we have records v_1, v_2, and v_3:

v_1: ANN 16 FEMALE 22 MAIN STREET

v_2: TOM 16 MALE 22 MAIN STREET

v_3: JOHN 30 N ELM ROAD

We can get the vectors of v_1, v_2, and v_3 as follows by the multidimensional keyword space:

v_1: 1 0 1 0 1 0 0 0 1 0 1 0 1 0

v_2: 0 0 0 0 1 1 0 0 1 1 1 0 1 0

v_3: 0 1 0 1 0 0 1 1 0 0 0 0 0 1

Then, the vectors of v_1, v_2, and v_3 are input into Function (1) to get their Hilbert numbers h_1, h_2, and h_3.

h_1 = H(1 0 1 0 1 0 0 0 1 0 1 0 1 0) = 6630

h_2 = H(0 0 0 0 1 1 0 0 1 1 1 0 1 0) = 6688

h_3 = H(0 1 0 1 0 0 1 1 0 0 0 0 0 1) = 16243

Thus, the ten-dimensional vectors are hashed to one-dimensional integers (i.e., Hilbert numbers). Because v_1 and v_2 have common keywords "16", "22", "MAIN" and "STREET", they are similar records. v_3's Hilbert number is not close to those of v_1 and v_2 because it does not have any keywords contained in v_1 and v_2, so v_3 is not as similar as v_1 and v_2. From the Hilbert numbers of v_1, v_2, and v_3, we notice that the difference of Hilbert numbers of similar records v_1 and v_2 is smaller than the difference of the Hilbert numbers of v_1 and v_3. This implies that v_2 is closer to v_1 than v_3. Consequently, close points have close Hilbert numbers, i.e., close data records can be clustered together based on their Hilbert numbers. To look for close records, we only need to check the closeness of the Hilbert numbers of records.

3.4 Hash-based Similar Records Clustering

Because a massive database has a huge number of records, it will take a long time to search close records by checking the Hilbert number of records one by one. Rather than reactively searching, we develop a database structure and searching algorithm to proactively handle close point queries. Specifically, we cluster the data records into different groups based on their closeness. That is, the records with the same Hilbert number are clustered into one group. The index of a source record is its location in the database, where the source record can be fetched. We divide a single record index database into a number of sub-databases, with each sub-database responsible for a record index group with high similarity, i.e., with the same Hilbert number. A centralized location index is used to record the location of each sub-database in the database and its responsible Hilbert number. A location index is the Hilbert number of a group of records in a sub-database.

To insert a data point in the record clustering model, the Hilbert number of the point is computed first - as an example, we use 5. Then the location index is referenced to get the location of the sub-database with Hilbert number equal to 5. If the location does not exist in the location indices, a new sub-database with the new index is generated, and the location of the sub-database is added into the location index. If location index 5 is in the location record, the data point will be directly stored in the sub-database pointed to by the location link. A sub-database, which is linked with corresponding hash ID in the location index, is constructed as a linked list in which all the records have the same Hilbert number. To store the data point in the sub-database, we only need to store the index of the data point at the end of the linked list.

Figure 5 shows the process of inserting a data point into the corresponding cluster. By using hash Function (1), data point v receives its Hilbert number h_v, which is 5. There is "5" in the location index, so data point v is stored in the sub-database linked with location index 5. This proactive data structure and clustering algorithm significantly reduces the searching cost by eliminating the need to go through the entire database reactively. A hash table (i.e., location index) is used for record clustering to save sub-databases. The Hilbert number of a source record, which is also location index, is the hash ID in the hash table. The links corresponding to hash IDs in the hash table

point to different sub-databases.

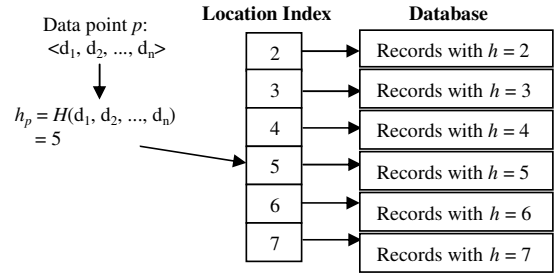


Figure 5: The process of record clustering.

3.5 HCS Similarity Searching Process

With the previously introduced VSM-based record vector generation method, the similarity between the vectors of two records remains the same as the similarity of the two records.

A token list has a very large number of unique keywords, so the dimension of a record vector is large. However, each record contains only a small number of keywords. Therefore, in the vector of a record, most positions are 0s. This leads to sparsity of the record vector. In a high dimensional space, a Hilbert curve is sensitive to the sparsity of the record vectors, and it may generate different Hilbert numbers for similar records. In order not to miss some records similar to the query, HCS uses multiple token lists to achieve high performance for similar information searching. HCS first generates m token lists. All unique keywords in the token lists are in random order. According to different token lists, a series of vectors is produced. Because there are m token lists, m vectors are made for each source record. Figure 6 shows the Hilbert numbers of a record according to different token lists. T_1, T_2, T_3 and T_4 are the token lists with different keyword orders. From Figure 6, we can see that different token lists lead to different Hilbert numbers for a record. HCS then uses the Hilbert hash function (1) to get m Hilbert numbers for each vector. Based on each of the m Hilbert numbers, the indices of source records are clustered and saved in each of the m databases.

When querying a record, HCS computes Hilbert numbers under the different token lists for the query record. Then, it checks each hash table accordingly, where the Hilbert number of a record is the hash ID in the hash table. With the hash ID, the sub-databases that have the same Hilbert numbers as the query record are easily located, and the records within these sub-databases are considered to be similar records of the query record.

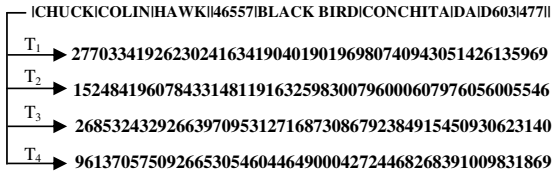


Figure 6: An example of Hilbert numbers from different token lists.

Figure 7 shows the process of record clustering and similarity searching in HCS. First, m token lists are generated. Second, according to different token lists, m groups of source record vectors are produced. Third, each vector is transformed into a Hilbert number using a Hilbert curve hash Function (1). Finally, the indices of the source records are saved in m hash tables. Each hash table stores the indices of source records according to the Hilbert numbers, which are computed from a token list. When searching for records similar to a query record q, m vectors of query q are produced based on the m token lists. HCS then uses hash function (1) to get m Hilbert numbers for query q. The m hash tables are checked one by one. The hash table i is checked according to the Hilbert number that is computed from token list i, $1 \leq i \leq m$. From Figure 7 we can see that query record q has the same hash ID as record v_2 in hash table 1 and it has the same hash ID as record v_3 in hash table m. Therefore, records v_2, v_3, and other records that have the same hash ID as query q in other hash tables are identified as similar records of q. Algorithm 1 shows the pseudo-code for record clustering and similarity searching in HCS.

A range can also be employed to enlarge the search scope. With the range r, the records with Hilbert number h_j that satisfy the condition $|h_j - h_q| \leq r$ are also checked, where h_q is the Hilbert number of a query record. Therefore, more similar records are located for the query record.

Let's use an example to explain the similarity searching process of HCS. There are four records in a database and the query record q is:

v_1: Ann Johnson 16 Female 248 Dickson Street

v_2: Ann Johnson 20 Female 168 Garland

v_3: Mike Smith 16 Male 1301 Hwy

v_4: John White 24 Male Fayetteville 72701
q: John White 20 Female 168 Garland

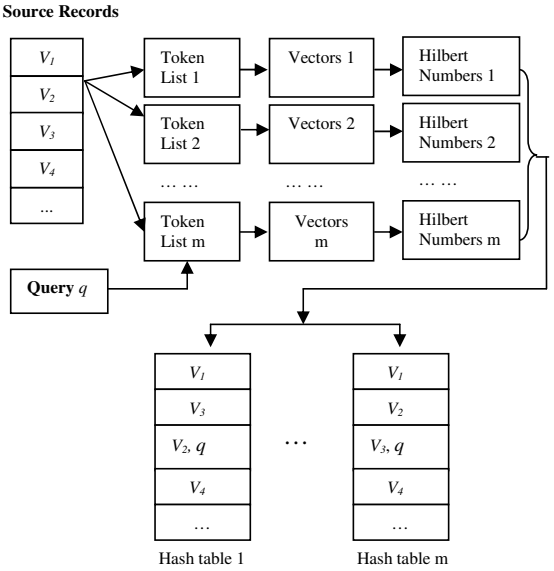


Figure 7: The process of record clustering and similarity searching in HCS.

We generate two token lists for this database. All the records are transformed into vectors and then Hilbert numbers, which are shown in Figure 8.

Token List 1		
Record	**Vector**	**Hilbert Number**
v_1	1001001001010011 0000	35953
v_2	1001000101001000 1000	123662
v_3	0100101000 1001000100	247708
v_4	0010010010000000011	525880
q	0010010101001000 1000	123704
Token List 2		
Record	**Vector**	**Hilbert Number**
v_1	1000100110100010 0010	493281
v_2	0101000100110010 0000	30476
v_3	1000010000001001 1100	188478
v_4	0010001001000101 0001	998520
q	0101000000 1101000001	1034252

Figure 8: Vectors and Hilbert numbers of records.

Because there are two token lists, two hash tables are used to save the record indices according to Hilbert numbers that are computed from different token lists. Hash table 1 saves the record indices according to the computation results from token list 1, and hash table 2 saves the record indices according to the computation result from token list 2. The two hash tables are presented in Figure 9. A record's hash ID is its location index in a sub-database that consists of the record indices linked with the hash ID. When searching for records similar to a query record, the hash tables are checked one by one. We apply the range query in the search, and range r is set to 50000. Assume the Hilbert number of q is 123704 for hash table 1 and 1034252 for hash table 2. When searching hash table 1, HCS checks the Hilbert numbers of source records h_j $(1 \leq j \leq 4)$. If they satisfy the condition $|h_j - 123704| \leq 5000$, the source records are considered to be similar to query q. Therefore, v_2 is retrieved as a similar record of query q. When searching hash table 2, record v_4 satisfies the condition $|h_j - 1034252| \leq 5000$. Therefore, v_4 is similar to query q. HCS combines the query results from hash table 1 and hash table 2, and determines that v_2 and v_4 are similar records of query q. Then, HCS calculates the Euclidean distance to measure the vector distances from located records and the query, and identifies the records whose distances are less than a predefined threshold as the final similar records.

Algorithm 1 Pseudo-code for HCS record clustering and similarity searching.

```
1: Generate m token lists token_list[1]...token_list[m]
2: for i=1 to m do
3:    for each record source[j] do
4:       Generate vector v[i][j] according to token_list[i]
5:       Calculate hashID[i][j] based on vector v[i][j]
6:       if hashID[i][j] does not exist in hash_table[i] then
7:          Save hashID[i][j] in hash_table[i]
8:       end if
9:       Save the index j of record source[j] in the corresponding place
         in hash_table[i]
10:   end for
11:   for each query record query[k] do
12:      Generate vector q[i][k] according to token_list[i]
13:      Calculate hashID[i][k] based on vector q[i][k]
14:      if hashID[i][k] exists in hash_table[i] then
15:         Save all the indices linked with hashID[i][k] into result[k][i]
16:      else
17:         Save null into result[k][i]
18:      end if
19:   end for
20: end for
21: Unite all the results from the result[k][i]
```

Analysis of HCS. Multiple token lists are used in HCS in order to improve the search performance. The more token lists, the fewer similar records that will be missed. One question that arises is what percentage of similar records can be located with a certain number of token lists? We assume p is the percentage of similar records

	Hash Table 1	
	Hash ID	**Record**
	35953	v_1
q →	123662	v_2
	247708	v_3
	525880	v_4
	Hash Table 2	
	Hash ID	**Record**
	30476	v_2
	188478	v_3
	493281	v_1
q →	998520	v_4

Figure 9: Hash tables.

located by using one token list; then, $q = 1 - p$ is the percentage of similar records not located by using one token list. Let $F(m)$ denote the percentage of similar records that can be located using m token lists. Then, we can get $F(n) = 1 - q^m$. Since $q = 1 - p$,

$$F(m) = 1 - (1 - p)^m. \quad (2)$$

Function (2) will help us to find the percentage of similar records located by different number of token lists. For example, if p equals 0.2 and m equals 5, $F(m)$ equals 0.67232. This means HCS can locate 20% of similar records using one token list and can locate approximately 67.23% of similar records using 5 token lists.

4 Performance Evaluation

We implemented HCS and conducted the comparison between HCS schemes with different numbers of token lists (HCS-k, where k represents the number of token lists) and the linear search method. We used two data sets in the experiments. Dataset 1 has 10,000 source records and 33,601 unique keywords (i.e., the dimension of the dataset is 33,601). We randomly chose 97 records as query records. Dataset 2 has 34,513 source records and 62,223 unique keywords. We randomly chose 10,000 queries as query records. Unless otherwise specified, we used dataset 1 in the test.

Because of the high dimensionality of the space, any Hilbert numbers are huge real numbers, and the difference of the Hilbert numbers of close records is a huge number. As an optimization, instead of using the entire Hilbert number, we only use the first L digits as the Hilbert number of the record. For example, two records' whole Hilbert numbers are 2348910847362 and 2348994736208, so the first 5 digits, 23489, are used as the new Hilbert number. Two records with the same Hilbert number are considered to be similar. In the experiments, we used the first 53 digits of the entire Hilbert number as the Hilbert number of the record, and we regard a record with at least one keyword in common with the query as a similar record, unless otherwise specified.

The metrics we tested are:

- *Total query time*. This shows the efficiency of a similar information searching method in terms of search latency.
- *Memory consumption*. This shows the efficiency of a similar information searching method in terms of memory required.
- *Effectiveness*. This represents the number of true positives and false positives returned in the located similar records. A true positive is a located record which is actually similar to the query record. A false positive is a located record which is not similar to the query record. High effectiveness means that a similar information searching method can locate similar information more accurately.
- *The scope of retrieved similar records*. This shows whether a similar information searching method can locate similar records with different similarities to the query record.
- *The number of true positives with range R*. This shows the number of true positives located with different R ranges.

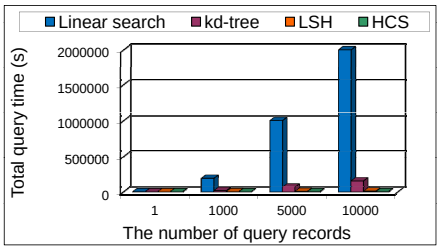


Figure 10: Total query time of different similar information search schemes.

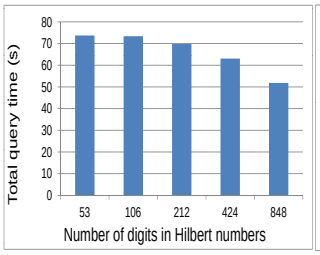


Figure 11: The total query time versus Hilbert number length.

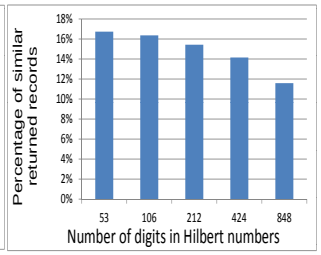


Figure 12: Percentage of similar records versus Hilbert number length.

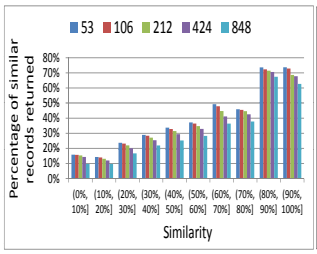


Figure 13: The percentage of returned similar records with different similarities for Dataset 2.

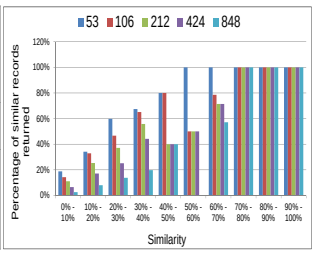


Figure 14: The percentage of returned similar records with different similarities for Dataset 1.

4.1 Comparison of Query Times of Different Schemes

Figure 10 shows the total query time of the linear search method, kd-tree search method, LSH search method and HCS. We see that the query time follows Linear>kd-tree>LSH>HCS. The linear search method needs to compare every record with the query, leading to a much higher query latency. The kd-tree search method and LSH search method reduce the query latency using the kd-tree structure and the LSH hash function family. HCS only needs to hash the query once and then checks the mapped cluster to find similar records, leading to the least query latency.

4.2 Effect of the Number of Digits in Hilbert Numbers

For this experiment, we used dataset 2. Figure 11 and Figure 12 show the total query time and the percentage of similar returned records versus the number of digits of Hilbert numbers, respectively. From Figure 11, we see that shorter Hilbert number lengths leads to higher total query times and vice versa. This is because shorter Hilbert number lengths cause more records to have the same Hilbert number after the mod operation, thus requiring more filtering time to derive actual similar records. From Figure 12, we see that the shorter lengths of Hilbert numbers leads to higher percentages of similar returned records and vice versa. Since shorter lengths cause more records have the same Hilbert number, a greater number of similar records are returned.

Figure 13 shows the percentages of returned similar records with different similarities to the query using Dataset 2. We see that shorter Hilbert numbers have higher probabilities of locating similar records than longer Hilbert numbers due to the reasons explained previously. Also, records with higher similarity to the query are more easily found regardless of the Hilbert number length. Figure 14 shows the percentages of returned similar records with different similarities to the query using Dataset 1. We have the same observations as in Figure 13.

Table 1: Accuracy.

Similarity	53 digits	106 digits	212 digits	424 digits	848 digits
1.0	Y	Y	Y	Y	Y
0.9	Y	Y	Y	Y	Y
0.8	Y	Y	Y	Y	Y
0.7	Y	Y	N	N	N
0.6	Y	N	N	N	N
0.5	Y	N	N	N	N
0.4	Y	N	N	N	N
0.3	N	N	N	N	N
0.2	N	N	N	N	N
0.1	N	N	N	N	N

We define *accuracy rate* as the total number of located similar records divided by the total number of existing records. Table 1 shows the accuracy rate for different Hilbert number lengths. We see that when more digits are used to represent the Hilbert number of records, fewer similar records are found and the records with higher similarity can be easily found. This is due to the same reasons as in Figure 12.

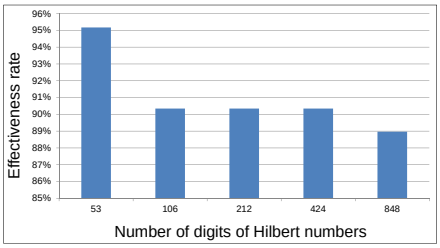


Figure 15: Effectiveness rate with different Hilbert number lengths used.

We define the *effectiveness rate* as the percentage of

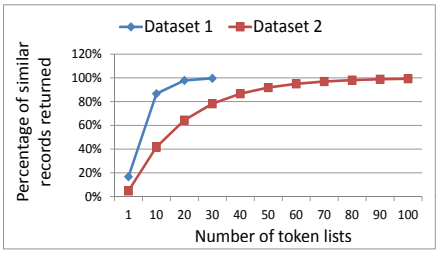


Figure 16: The percentage of returned similar records.

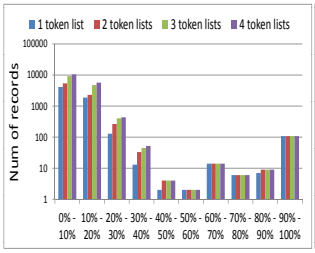


Figure 17: Number of similar records returned in Dataset 1.

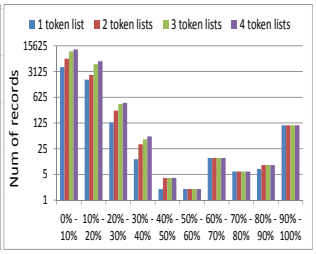


Figure 18: Number of similar records returned in Dataset 2.

the located similar records in the returned records. Figure 15 shows the effectiveness rate for different lengths of Hilbert numbers. We see that longer Hilbert number lengths lead to lower effectiveness rates. This is because a longer length produces a finer granularity of data, thus generating fewer returned similar records.

4.3 Effect of the Number of Token Lists

Figure 16 plots the percentage of returned similar records using Dataset 1 and Dataset 2. The figure shows that more token lists help to find more similar records. When the number of token lists reaches a certain value, a further increase in the token lists leads to a slight increase in the percentage of similar returned records. Dataset 1 needs fewer token lists to locate all similar records due to the data set's smaller dimension. This experimental result shows that the number of token lists needed to locate almost all similar records varies based on the dimensions of data sets.

Figure 17 and Figure 18 show the numbers of similar records returned in Dataset 1 and Dataset 2, respectively. Figure 19 shows the percentage of returned similar records with different similarities. We see that the records with high similarity with the query can be found regardless of the number of token lists. However, more

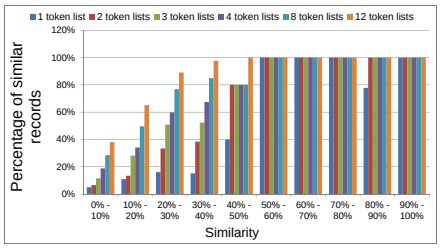


Figure 19: The percentage of returned similar records with different similarity.

token lists are needed to find the records less similar to the query. Also, we can observe that by adding more token lists, the number of similar records returned becomes greater in each similarity area. One token list can almost locate records with more than 50% similarity; 12 token lists can locate records with more than 40% similarity. With more token lists, the records with lower similarities to the query can be more easily found.

Figure 20 shows the total query latencies of different methods. In the figure, HCS-m means HCS with m token lists. The query speed of HCS is much faster than linear search. HCS clusters the similar records first, enabling the query to be directly be mapped to specific clusters to instead of searching the entire database. In contrast, the liner search method searches the entire database and compares each source record to the query record in order to find the similar records, leading to a much higher querying latency. We also observe that HCS using more token lists produces a higher query latency. This is because when using more token lists in HCS, the time for querying similar records increases. The increase in the number of token lists leads to the increase in the number of hash tables for storing the clustered source record indices. Then, more hash tables should be checked to query the similar records. From Figure 20, we also see that when the number of token lists increases by one, the query time increases about 0.03 seconds. Therefore, approximately 0.03 seconds are needed for searching one hash table.

Figure 21 shows the total query time versus the number of token lists in HCS. It illustrates that the total query time increases almost linearly as the number of token lists increases. This is because adding one more token list means one more token list needs to be checked during data search.

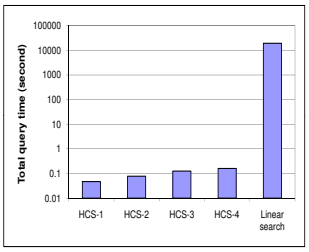


Figure 20: Latency of HCS and the linear search method.

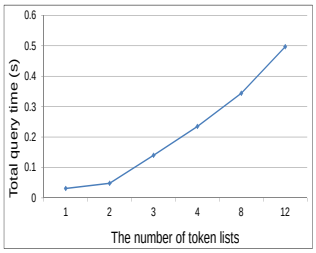


Figure 21: Latency versus the number of tokens in HCS.

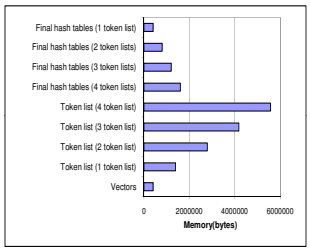


Figure 22: Memory consumption.

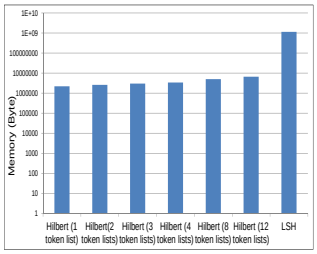


Figure 23: Memory consumption.

HCS needs memory for saving token lists, the vectors of records, and the final hash tables. Figure 22 presents the memory consumption of different parts of HCS. The memory for storing vectors can be reused to save vectors for different token lists after hashing the source records into a hash table. For instance, HCS requires memory for saving the vectors that are produced according to a token list. After hashing the records to a hash table, HCS continues to generate another token list and produces another vector list. Because the vectors of the first token list will not be used subsequently, the memory for storing the vectors of the previous token list can be used for the new vectors. Therefore, the memory consumption required for storing vectors does not change in the different HCS methods and increases as the number of keywords increases. From Figure 22, we can observe that the memory consumed for saving token lists and final hash tables increases when more token lists are used. When one more token list is used, one more final hash table is required to save the source records. Therefore, the memory required for saving the token list and final hash tables increases. When the number of token lists increases by one, 1,400,000 bytes are required for storing the token list and 400,000 additional bytes are used for storing the final hash tables. Figure 23 shows the memory consumption of HCS with different numbers of token lists. We see that the memory consumption increases as the number of token lists increases since more hash tables need more memory space. HCS with 12 token lists still consumes much less memory than LSH.

Figure 24 presents the total number of located records including true positives and false positives in different methods. When HCS employs more token lists, it can locate more similar records. Using one token list, HCS can locate about 5% of the actual similar records; using twelve token lists, HCS can locate about 40% of the similar records. However, as more similar records were located, more false positive were concurrently generated. The percentage of false positives in the located records is much lower than the percentage of true positives. Therefore, increasing the number of token lists can help to find more similar records, with the side-effect of returning more false positives. We also see that HCS finds fewer similar records than the linear search method. More token lists enable HCS to find more similar records.

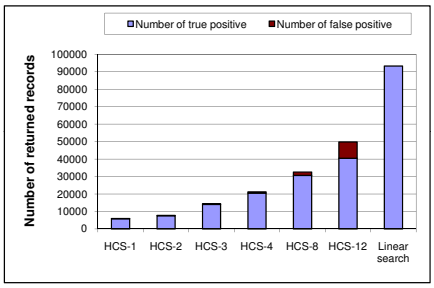


Figure 24: The total number of located records for HCS and linear search.

In order to see the degree of similarity located records have to the query record in HCS, we conducted experiments on HCS with 1, 2, 3, and 4 token lists. We randomly chose one record, and changed one token to make a new record for the query each time with the aim of determining if HCS can still find the original record with the decreasing degree of similarity to the query record. Table 1 shows whether the methods can find the original record when the record has different similarities to the query record. In the table, "Y" and "N" mean that the method can and cannot find the original record, respectively. Given two records A and B, their similarity is calculated with the following

Table 2: The scope of retrieved similar record.

Similarity	HCS-1	HCS-2	HCS-3	HCS-4
1.0	Y	Y	Y	Y
0.9	Y	Y	Y	Y
0.8	N	Y	Y	Y
0.7	N	Y	Y	Y
0.6	N	Y	Y	Y
0.5	N	N	Y	Y
0.4	N	N	Y	Y
0.3	N	N	N	N
0.2	N	N	N	N
0.1	N	N	N	N

function:

$$Similarity = \frac{|A \cap B|}{|A|}. \quad (3)$$

Table 2 illustrates that HCS can locate the records with low similarity when more token lists are used. HCS with 3 and 4 token lists can locate the records whose similarities to the query record are greater than 0.3. HCS with more token lists is able to locate records with low similarity because it generates more Hilbert numbers for each record has a large scope of possible Hilbert numbers that can be checked. The results imply that records having higher similarities to the query record have a higher probability of being located than records having lower similarities. Multiple token lists should be used to locate the similar records with low similarity.

4.4 The Number of Token Lists Needed for an Expected Percentage of True Positives

Since increasing the number of token lists can locate more similar records, we want to know how many token lists are needed for locating all the similar records. We conducted another simulation with the help of Function (2). In Function (2), we set the value of p to the percentage of similar records located by using one token list. As the value of m increases, a higher percentage of similar records will be located. Figure 25 shows the expected percentage of true positives and the percentage of true positives versus the different number of token lists. The number that is calculated by Function (2) is named "Expected percentage of true positives"; "Percentage of true positives" denotes the actual experimental result. The figure indicates the number of token lists needed for locating a certain percentage of similar records. Figure 26 plots the number of true positives versus the number of token lists. From the figures, we notice that the percentage of true positives is consistent with the expected percentage of true positives. We also notice that with one token list, HCS can locate about 5% of the similar records, and 100 token lists are needed for locating more than 99% of the similar records. Considering the results of search latency and memory consumption described above, we can determine that the total query time of HCS with 100 token lists is about 3 seconds, which is still much faster than the linear search method. The memory needed to store the 100 final hash tables is 40,000,000 bytes.

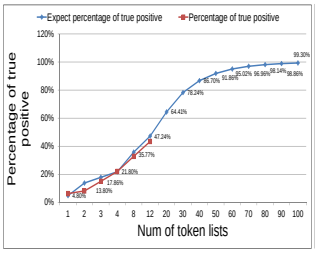


Figure 25: The number of located similar records for different values of R.

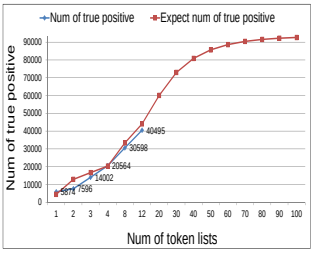


Figure 26: The number of token lists needed to locate a certain number of true positives in our dataset.

4.5 The Effect of Searching Scope R

In addition to checking the hash tables at the exact location index of a query record, near neighbours in the hash tables are also checked in our experiment. For example, if a query record's hash index is 10 and the range R for checking near neighbours is 2, then we collect all the records saved in locations 8, 9, 10, 11, and 12. This near neighbour query increases the searching range, which can locate the points that are not very close to the query point. Figure 27 shows the number of located similar records with different values of R. Figure 28 shows the results without the linear search method. From the figures, we can see that the number of located similar records increases as the value of R increases. A larger R can help to locate more similar records. Increasing the value of R means more grids can be checked in high dimensional space, and more points fall in the checking area. This increases the probability of finding more similar records, because similar records are close to each other and the differences between their Hilbert numbers are small.

Figure 29 shows the percentage of returned similar records versus the searching scope R. We see that as the

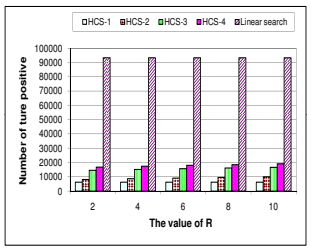


Figure 27: The number of token lists needed for locating a certain percentage of true positives.

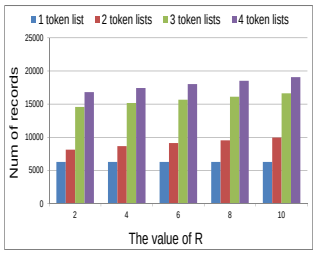


Figure 28: The number of located similar records for different values of R.

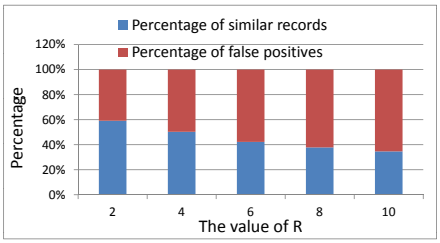


Figure 29: The percentage of returned similar records versus R.

value of R increases, the percentage of similar records decreases, while the percentage of false positives increases. Larger R values generate more record candidates to check for similar records to the query. Thus, more false positives are introduced, and hence the percentage of true positives is reduced. The result implies that an appropriate R should be chosen to increase the true positives while minimizing the false positives.

5 Conclusions

This paper proposes a Hilbert curve based similarity searching scheme (HCS). HCS utilizes the Hilbert curve's locality preserving property to effectively group similar records. HCS treats the records in databases as the points in a high-dimensional space. It uses a vector to present each point. A Hilbert curve is used to project points from a multidimensional space to an one-dimensional space. Therefore, the multidimensional vectors of points can be represented as a single integer number called a Hilbert number. Hilbert numbers can reflect the closeness of two records. Finally, the records are saved in a hash table according to their Hilbert numbers. This process classifies the records into a cluster based on their closeness (i.e., similarity). A query record is also assigned a Hilbert number that can map the query to a cluster. Comparison is conducted between the query record and the records in the cluster, and the similar records are returned. We further propose HCS with multiple multidimensional spaces (i.e., token lists) to improve the similarity searching performance. Simulation results show the superior performance of HCS compared to the linear search algorithm in terms of query latency. HCS dramatically reduces the query time and exhibits high effectiveness in desired information retrieval. In our future work, we will investigate how to increase true positives and reduce false positives of HCS in the similarity searching in a massive database.

Acknowledgements

This research was supported in part by U.S. NSF grants OCI-1064230, CNS-1049947, CNS-1156875, CNS-0917056 and CNS-1057530, CNS-1025652, CNS-0938189, CSR-2008826, CSR-2008827, Microsoft Research Faculty Fellowship 8300751, and Oak Ridge Award 4000111689.

References

[1] D. J. Abel and D. M. Mark. A comparative analysis of some two-dimensional orderings. *International Journal of Geographical Information Science*, 4(1), January 1990.

[2] C. C. Aggarwal. Hierarchical subspace sampling: A unified framework for high dimensional data reduction, selectivity estimation and nearest neighbor search. In *Proceedings of ACM SIGMOD Conference*, 2002.

[3] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9), 1975.

[5] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *Proceedings of the 7th International Conference on Database Theory (ICDT)*, 1999.

[6] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4), 1973.

[7] J. Castro, M. Georgiopoulos, R. Demara, , and A. Gonzalez. Data-partitioning using the hilbert space filling curves:

Effect on the speed of convergence of fuzzy artmap for large database problems. *Neural Networks*, 18(7), September 2005.

[8] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3), 2001.

[9] P. Ciaccia, M. Patella, and P. Zezula. M-trees: an efficient access method for similarity search in metric space. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, August 26-29, 1997.

[10] T. Darrell, P. Indyk, and G. Shakhnarovich (eds.). *Nearest Neighbor Methods in Learning and Vision: Theory and Practice*. MIT Press, Erewhon, NC, 2006.

[11] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry (SCG)*, 2004.

[12] C. Digout. Metric techniques for high-dimensional indexing. Technical Report TR 04-19, University of Alberta, Canada, September 2004.

[13] C. Digout and M. A. Nascimento. High-dimensional similarity searches using a metric pseudo-grid. In *Proceedings of the 21st International Conference on Data Engineering Workshops (ICDEW)*, 2005.

[14] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD)*, 2003.

[15] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, , and W. Equitz. Efficient and effective querying by image content. *Intelligent Information Systems*, 3(3-4):231–262, July 1994.

[16] A. Fu, P. M. S. Chan, Y. L. Cheung, and Y. S. Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *VLDB*, 9(2), 2000.

[17] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, 1999.

[18] D. Grossman and O. Frieder. *Information Retrieval: Algorithm and Heuristics*. Springer, Netherlands, 2004.

[19] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of International Conference On Management Of Data*, pages 47–57. ACM, 1984.

[20] J. J. Barthholdi III and L. K. Platzman. Heuristics based on spacefilling curves for combinatorial problems in euclidean space. *Manage. Sci.*, 34(3), 1988.

[21] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of 13th Annual ACM Symposiumon Theory of Computing*, 1998.

[22] III J. J. Bartholdi and P. Goldsman. Vertex-labeling algorithms for the hilbert spacingfilling curve. *Software Practice and Experience*, 31(5), 2001.

[23] H. V. Jagadish. Linear clustering of objects with multiple attributes. *SIGMOD Rec.*, 19(2), May 1990.

[24] N. Katayama and S. Satoh. The sr-tree: an index structure for high-dimensional nearest neighbor queries. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data (SIGMOD)*, 1997.

[25] John L. Kelley. *General Topology.* Springer-Verlag, 1975.

[26] M. Kppen. The curse of dimensionality. http://www.npt.nuwc.navy.mil/Csf/papers/hidim.pdf.

[27] S. Kulkarni and R. Orlandic. *High-dimensional similarity search using data sensitive space partitioning*, volume 4080/2006. Springer Berlin/Heidelberg.

[28] N. Linial and O. Sasson. Non-expansive hashing. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing (STOC)*, 1996.

[29] X. Lu, Y. Wang, and A. K. Jain. Combining classifiers for face recognition. In *Proceedings of the 2003 International Conference on Multimedia and Expo - Volume 3 (ICME)*, 2003.

[30] D. Maio and D. Maltoni. A structural approach to fingerprint classification. In *Proceedings of the International Conference on Pattern Recognition (ICPR)*, 1996.

[31] Clemens Marschner. Mtree tester applet. http://www.cmarschner.net/mtree.html.

[32] M. Patella and P. Ciaccia. The many facets of approximate similarity search. In *Proceedings of the First International Workshop on Similarity Search and Applications (SISAP)*, 2008.

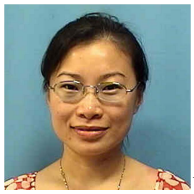
[33] H. Sagan. *Space-Filling Curves*. Springer-Verlag, New York, 1994.

[34] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. Technical report, Cornell University, Ithaca, NY, USA, 1974.

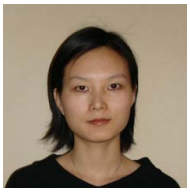
[35] S. Santini and R. Jain. Beyond query by example. In *Proceedings of the sixth ACM international conference on Multimedia (Multimedia)*, 1998.

[36] R. Weber, H.-J Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB)*, 1998.

[37] D. A. White and R. Jain. Similarity indexing with the ss-tree. In *Proceedings of the Twelfth International Conference on Data Engineering (ICDE)*, 1996.



[ ]Ting Li Ting Li received the BS degree in Electronics and Information Engineering from Huazhong University of Science and Technology, China, in 2007. He is currently a Ph.D. student in the Department of Electrical and Computer Engineering of Clemson University. His research interests include distributed networks, with an emphasis on peer-to-peer and content delivery networks, wireless multi-hop cellular networks, game theory and data mining. He is a student member of IEEE.



[ ]Haiying Shen Haiying Shen received the BS degree in Computer Science and Engineering from Tongji University, China in 2000, and the MS and Ph.D. degrees in Computer Engineering from Wayne State University in 2004 and 2006, respectively. She is currently an Assistant Professor in the Department of Electrical and Computer Engineering at Clemson University. Her research interests include distributed computer systems and computer networks, with an emphasis on P2P and content delivery networks, mobile computing, wireless sensor networks, and grid and cloud computing. She was the Program Co-Chair for a number of international conferences and member of the Program Committees of many leading conferences. She is a Microsoft Faculty Fellow of 2010 and a member of the IEEE and ACM.