J Intell Inf Syst
DOI 10.1007/s10844-013-0289-9

Token list based information search in a multi-dimensional massive database

Haiying Shen · Ze Li · Ting Li

Received: 27 March 2013 / Revised: 6 November 2013 / Accepted: 8 November 2013

Abstract Finding proximity information is crucial for massive database search. Locality Sensitive Hashing (LSH) is a method for finding nearest neighbors of a query point in a high-dimensional space. It classifies high-dimensional data according to data similarity. However, the "curse of dimensionality" makes LSH insufficiently effective in finding similar data and insufficiently efficient in terms of memory resources and search delays. The contribution of this work is threefold. First, we study a Token List based information Search scheme (TLS) as an alternative to LSH. TLS builds a token list table containing all the unique tokens from the database, and clusters data records having the same token together in one group. Querying is conducted in a small number of groups of relevant data records instead of searching the entire database. Second, in order to decrease the searching time of the token list, we further propose the Optimized Token list based Search schemes (OTS) based on index-tree and hash table structures. An index-tree structure orders the tokens in the token list and constructs an index table based on the tokens. Searching the token list starts from the entry of the token list supplied by the index table. A hash table structure assigns a hash ID to each token. A query token can be directly located in the token list according to its hash ID. Third, since a single-token based method leads to high overhead in the results refinement given a required similarity, we further investigate how a Multi-Token List Search scheme (MTLS) improves the performance of database proximity search. We conducted experiments on the LSH-based searching scheme, TLS, OTS, and MTLS using a massive customer data integration database. The comparison experimental results show that TLS is more efficient than an LSH-based searching scheme, and OTS improves the search

H. Shen (✉)
Department of Electrical and Computer Engineering, Clemson University, Clemson, SC 29634, USA
e-mail: shenh@clemson.edu

Z. Li
MicroStrategy, Tysons Corner, Fairfax, VA 22182, USA
e-mail: zel@clemson.edu

T. Li
Wal-mart Stores Inc., Bentonville, AR, 72716, USA
e-mail: dragonflyting@hotmail.com

efficiency of TLS. Further, MTLS per forms better than TLS when the number of tokens is appropriately chosen, and a two-token adjacent token list achieves the shortest query delay in our testing dataset.

Keywords Similarity data search · Proximity search · Locality sensitive hash · Database

1 Introduction

A database of complex objects aims at storing such objects and providing a means to access them according to their content (Loccoz 2005). With the rapid growth of information, efficient information searching of massive databases is becoming important. Besides finding exact information, people have increasing interest in looking for similar information that contains certain keywords.

For example, in the eBay website, the search engine returns not only the products a buyer desires to buy according to the entered keywords, but also returns the goods relevant to the keywords that the buyer might be interested in. In the bioinformatics field, since completely identical DNA sequences are difficult to find, scientists are more interested in identifying similar DNA sequences in research. Even in the image and speech processing fields, image and speech documents are represented by high-dimensional color histograms and a large number of coefficients (Cover and Hart 1967; Deerwester et al. 1990; Fagin 1998). Therefore, similarity information search offers users more choices and resources.

In this paper, we consider similarity information search with a focus on keyword search in a multi-dimensional massive database. Such a database consists of a large number of data records and each record is described by a number of tokens such as customer name, address, gender, age in a customer dataset. Figure 1 shows an example for data records for customer data consisting of tokens such as “TOM” and “SMITH”. “TOM” and “SMITH” are the first two tokens of the first data record. All tokens in the system form a multi-dimensional token space, where each token is a coordinate. The number of dimensions equals the number of different tokens in the database, with each token representing a dimension.

Thus, the data records of a massive database are treated as points in the high-dimensional token space. Figure 2 shows an example of a 3-dimensional token space. Each data record is viewed as base-*m* vectors, where *m* is the total number of tokens in the system. Figure 3 demonstrates how a data record’s vector is generated. Given a data record, if it contains the token in a dimension, it has “1” in this dimension; otherwise, it has “0” in this dimension. As a result, similar records with more the same tokens are located closer in this high-dimensional space. Similar information search is to find records that have more the same tokens with a given query. As shown in Fig. 2, two data records are considered similar if they are adjacent in the token space.

Fig. 1 A dataset

#1	TOM SMITH 17 N ELM ST
#2	DAVID RUFF 22 MAIN ST
#3	....

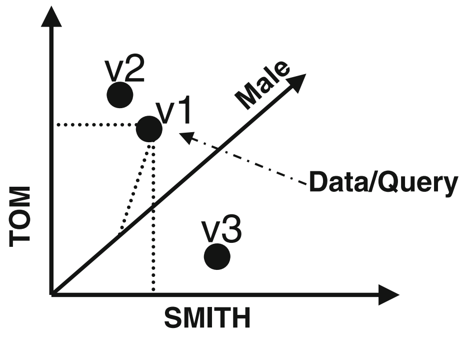
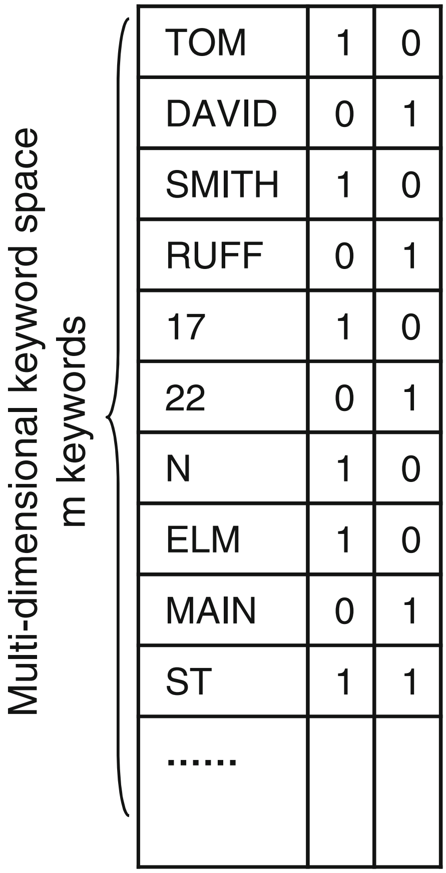


Fig. 2 A 3-D space

Thus, similar information search is also known as nearest neighbor search, which finds the nearest neighbor of a query in a high-dimensional space. The closest point of a query point is the nearest neighbor of the query point.

To retrieve the desired results efficiently, many different similarity/proximity information search methods have been proposed. The traditional linear search (Hu et al. 2005) method, as a brute force algorithm, compares a query record with every record in the database one


Multi-dimensional keyword space
m keywords

TOM	1	0
DAVID	0	1
SMITH	1	0
RUFF	0	1
17	1	0
22	0	1
N	1	0
ELM	1	0
MAIN	0	1
ST	1	1
......		

Fig. 3 A list of dimensions

by one. This method is not efficient in a massive database with complexity of $O(n)$ for a single query, where n is the number of data records in the database.

Other search methods (White and Jain 1996; Bentle et al. 1977; Arya et al. 1994; Niblack et al. 1993; Kruskal and Wish 1978; Panigrahy 2006) rely on a tree structure (e.g., kd-tree, BDD-tree, R-tree and LSD-tree and vp-tree) to hierarchically partition the database so that the search algorithm may quickly filter out the regions that do not overlap with the query. However, the tree-based methods require substantial space and time (Bohm et al. 2001; Sellis et al. 1997), and sometimes are less efficient than the linear searching approach (Hu et al. 2005). It has been shown that in a high-dimensional data space (>20), these methods tend to perform worse than the linear search method due to the curse of dimensionality, i.e., either the running time or the space requirement grows exponentially with dimension (Panigrahy 2006).

Clustering-based access methods (Bennett et al. 1999; Li et al. 2002; Berrani et al. 2003; Kleinberg 1997; Gionis et al. 1999) group the data points and access only the clusters that are likely to contain the nearest neighbors of the query point. While such methods may be efficient in terms of query time, accurately clustering similar data records is a challenge. Furthermore, clustering a very large database generates considerable cost. For example, locality sensitive hashing (LSH) is a known method that works faster than linear search for finding exact and nearest neighbors for high-dimensional data (Datar et al. 2004). Indyk et al. designed an LSH scheme based on p-stable distributions (Indyk and Motwani 1998). LSH can group records according to their similarity, which makes query procedure conducted in a group of similar data records instead of the entire database. The LSH scheme can find the exact near neighbors in $O(\log n)$ time, and the data structure is up to 40 times faster than the kd-tree searching scheme (Datar et al. 2004). However, it has drawbacks including a high memory requirement, long refinement processing time, and a same dimension requirement for input data records.

This paper presents a method - Token List based Searching scheme (TLS) - that improves on the LSH based searching scheme. TLS builds a token list to collect all the unique tokens in the records. Then, TLS maps each record to the token list according to its component tokens, and records the record's index at each token in the list. In searching, TLS maps a query to the token list based on the query record's token and retrieves all the record's indices stored in it. By classifying records base on token list, TLS provides efficient data searching. It shortens query time, reduces memory consumption, and works for different dimensional datasets.

In TLS, since each token in the token list records the indices of all the records that have it as a keyword, a record having a considerable number of keywords will take a long time in the refinement process to order the results based on their relevance to the query. Meanwhile, although the single-token based token list search can retrieve all the records sharing even one common keyword, it is not likely that users would be interested in these records in a real application. In order to reduce the token list searching time and final result refinement time, we further propose the Optimized Token list based Search schemes (OTS) based on index-tree and hash table structures. An index-tree structure orders the tokens in the token list and constructs an index table based on the tokens. Searching of the token list can be started from the entry of the token list supplied by the index table. Therefore, only part of a token list has to be searched to locate the query token. A hash table structure assigns a hash ID to each token. When searching for a token, a hash ID of the token can be calculated by a certain hash function, and the query token can be directly located in the token list according to the hash ID.

Since the single-token based method leads to a high overhead during results refinement given a required similarity, we further investigate how a multi-token list search scheme (MTLS) improves the performance of database proximity search in a multi-dimensional massive database. The multi-token based token list is an ordered list in which each entry stores multiple tokens for one record, as opposed to a single token. In this paper, *n*-token or multi-token based token list refers to the token list in which each entry contains several tokens of each record.

We have conducted experiments to analyze the potential of TLS, OTS, and MTLS for similarity information searching in a synthetic massive customer data integration (CDI) database. The experimental study shows that compared with LSH-based searching scheme, TLS is more effective in searching proximity information and requires much less memory and time in a massive database. Our experimental results also show that OTS is more efficient than TLS for searching latency. Further, the experimental results show that given a certain similarity requirement, MTLS achieves a higher performance than the single-token based token list. In our testing, the two-token based token list achieves the shortest query delay with our dataset.

Google indexing and search method (Blachman 2007) indexes web pages based on each keyword. In Google's index database, each index entry stores a list of documents containing the keyword. In web page searching, it finds the web pages containing each searching keyword, and finally combines the searched results as the searching results. Our basic TLS scheme also uses keywords to search data records containing each keyword, though our schemes aim to provide similar information search in multi-dimensional massive database, while Google indexing is for full-text database. However, our work is advantageous in that our OTS and MTLS schemes enhance the basic TLS scheme, which can be used by Google indexing and search method to further improve its searching performance.

The rest of this paper is structured as follows. Section 2 presents a concise review of representative methods for proximity information searching. Section 3 describes the design of TLS, OTS, and MTLS. Section 4 evaluates the performance of TLS, OTS, and MTLS in comparison with the LSH-based searching scheme. Section 5 concludes this paper with remarks on our future work.

2 Related work

In the past few years, there have been numerous studies on the problem of similarity search and finding the nearest neighbor of a query point in high-dimensional (at least three) space, focusing mainly on Euclidean space: given a database of n points in a d-dimensional space, find the nearest neighbor of a query point.

Linear searching (Hu et al. 2005) compares a query record with each record in the database one at a time. However, this method is inefficient since it leads to $O(n)$ time latency. Another approach uses distance information to deduce k-dimensional points for records so that the Vector Space Model multidimensional indexing method (White and Jain 1996) can be used. Examples of this method include the FastMap algorithm (Niblack et al. 1993) and Multidimensional Scaling (Kruskal and Wish 1978). Filho et al. (2001) proposed an Omni search strategy and introduced the concept of searching data based on some anchor points. The triangular inequality is used to limit the number of distance computations during searching.

Other similarity search methods can be classified into two categories: tree structure and dimension reduction clustering. Bentle et al. (1977) proposed a k-dimensional tree (kd-tree) data structure that is essentially a hierarchical decomposition of space with many dimensions. Kd-trees are effective in low dimensional spaces, but their searching performance degrades when the number of dimensions is larger than two. Panigrahy (2006) proposed an improved kd-tree search algorithm that iteratively perturbs the query point and traverses the tree. Balanced Box-Decomposition trees (BDD-trees) (Arya et al. 1994) are extensions of kd-trees with additional auxiliary data structures for approximate nearest neighbor searching. It recursively subdivides space into a collection of cells and measures the distance between a cell and a query point to determine whether the points in the cell should be options in the kd-tree search. These approaches map each record to a kd point and try to preserve the distances among the points. However, it is difficult to decide the value of k such that the mapping between each domain object to a k-dimensional point can accurately represent the similarity between objects.

A vantage point tree (vp-tree) (Fu et al. 2000) is a data structure that chooses vantage points to perform a spherical decomposition of the search space. This method is suited for non-Minkowski metrics and for lower dimensional objects embedded in a higher dimensional space (Yianlios 1993). The main drawback of vp-trees is that the region inside the median sphere and the region outside the median sphere are extremely asymmetric, and since volume grows rapidly as the radius of a sphere increases, the outside of the sphere tends to be very thin (Brin 1995).

R-tree (Rectangle-tree) (Guttman 1984) was proposed as an index structure for spatial searching. It is similar to a B-tree (Bayer and McCreight 1970; Comer 1979) with index records in its leaf nodes containing pointers to data objects. R-tree uses an n-dimensional rectangle to bind the data objects. A non-leaf node of the R-tree stores the address of a child node and a minimum bounding rectangle (MBR) of all entries within this child node (Kulkami and Orlandic 2006). The R-tree is efficient for data with a small number of dimensions but it shows poor performance for high-dimensional databases (dimensionality greater than 10) (Berchtold et al. 1996). R*-tree (Beckmann et al. 1990) is a variant of R-tree for indexing spatial information. R*-tree improves on the R-tree by not only minimizing the area of MBRs, but also minimizing the overlap of the MBRs and increasing the storage utilization of the MBRs (Beckmann et al. 1990).

Clustering methods have been proposed to accelerate the speed of similarity searches in a massive database. Clustering based similarity search methods aim at clustering the data points and accessing only the clusters that contain the nearest neighbors of the query point. Clindex (Li et al. 2002) is a clustering/indexing combined scheme for similarity search. Clindex divides the data space into a multi-dimensional grid and groups the neighboring cells that contain data points. It uses a cluster directory to keep track of information about all the clusters and a mapping table to map a cell to the cluster where the cell resides. When querying for a data point, Clindex maps the query point to a cell, and then looks in the mapping table to find the entry of the cell.

LSH is a method of performing probabilistic dimension reduction of high-dimensional data (Datar et al. 2004). It is used for resolving approximate and exact near neighbors in high-dimensional spaces (Datar et al. 2003; Indyk and Motwani 1998; Zolotarev 1986). It uses a special family of locality sensitive hash functions. The main idea of the LSH is to use the hash functions to hash high-dimensional points into a number of values, such that the points close to each other in their high-dimensional space will have similar hashed values. The points are classified based on their hashed values. Consequently, the near neighbors of a query point can be retrieved by locating the points with similar hashed values. Previous research on LSH in a p-stable distribution (Andoni and Indyk 2005) shows that LSH has a number of drawbacks. First, it requires a large amount of memory to achieve fast queries. Second, it needs refinement to filter false positives (located records not similar to the query record) out of the located records to achieve high accuracy, which leads to long processing latency. Third, LSH requires that the entire input dataset has the same dimensional identifier. LSH employs the vector model (Grossman and Frieder 2004) to get a record's identifier, which forms all tokens into a multidimensional keyword space where the tokens are the coordinates, and records are points in the space. A massive database has a tremendous number of tokens, and a record may contain only a few tokens. As a result, the identifier of a record which is transformed by the vector model has a lot of 0s and only a few 1s. This sparsity leads to high memory consumption and long refinement time.

In recent years, many information search methods have been proposed for large-scale distributed systems. Chen et al. (2008) proposed a multi-keyword search mechanism over peer-to-peer (P2P) Web environments. They aimed to obtain a minimal communication cost by tuning Bloom Filter settings and designed optimal order strategies for both "and" and "or" queries. The authors (Chen et al. 2010) also found that queries are typically short and users have limited interests, and proposed TSS that prunes term-set-based index by solely publishing the most relevant term sets from documents on the peers. TSS can reduce searching traffic cost while preserving comparable accuracy. Chaudhuri et al. (2007) introduced a cost model where keywords were assigned to three tiers based on document frequency; the most frequent keywords require extensive indexing while low-frequency keywords only need standard inverted indexes. This model can improve latencies in a multi-dimensional index environment. Lam et al. (2009) proposed an inverted index compression technique by pairing the posting lists of highly correlated terms. This approach can improve the compression ratio and save the size of posting cache. Long and Suel (2005) exploited frequently occurring pairs of terms and used an extra level of caching to cache intersections or projections of the corresponding inverted lists. AlvisP2P (Luu et al. 2008) allows peers to publish their local index and maintain a fraction of a global P2P index, which can improve network-wide accessibility of the local documents via the global search facility. Skobeltsyn et al. (2009) proposed a query-driven indexing framework for scalable text retrieval over structured P2P networks. The framework stores only top-k ranked document references and some carefully chosen term sets, thus reducing bandwidth consumption. Weth and Datta (2012) developed a query-driven mechanism to store popular term combinations derived from recent query history, and this multi-term indexing technique can support complex queries in a large-scale distributed storage systems.

Recall that this work has three components: (1) Token list based Search scheme (TLS), (2) optimized Token list based Search schemes (OTS), and (3) multi-token list search scheme (MTLS). Parts of the results on components (1) and (3) were initially published in the Proceedings of DMIN'08 (Li et al. 2008) and ICCIT'08 (Shen et al. 2008). In this paper, the second component include newly proposed methods for optimization: TLS with index table and TLS with hash table. We have conducted experiments to evaluate the component (2) compared with other components, and conducted additional experiments to evaluate other two components. We also have extended the description of component (1) and component (2) and all other sections in this paper with much more details.

3 Token list based searching scheme

3.1 The basic token list based searching scheme

Token List based Searching scheme (TLS) is designed for finding proximity information in a massive database.

Definition 1 A record is a string array that consists of a series of keywords. Each keyword is called a token or single-token.

For example, a record
Ann | Johnson | 16 | Female | 248 | Dickson | Street
consists of tokens "Ann", "Johnson", "16", "Female", "248", "Dickson" and "Street".

TLS builds a token list to collect all the unique tokens from the records in the database. Then, TLS maps each record to the token list according to its component tokens, and records the record's index at the entry of each of these tokens in the token list. In searching, TLS maps a query to the token list based on the query record's tokens, and retrieves all indices of records having one of the query's tokens. By classifying records based on a token list, TLS can shorten query time, reduce memory consumption, and work for different dimensional datasets. Figures 4 and 5 show the process of TLS. TLS reads the records in the database, and abstracts unique tokens in the records to build a token list. At the same time, TLS saves the indices of records in the token list according to their component tokens. To perform a query, TLS directly searches the entries with the query's tokens in the token list. The records whose indices are linked with query record's tokens are returned as the similar records of the query record. Specifically, TLS has two main steps: (1) constructing the token list and (2) searching the token list. In the following, we introduce the details of each step followed by an enhanced method to improve the performance of TLS.

3.1.1 Constructing token list

LSH builds a multi-dimensional space with each token representing a dimension and then assigns a vector (identifier) to each data record. If the data record has the token, it has 1 in the token's dimension; otherwise, it has 0 in the token's dimension. Unlike LSH, TLS does not

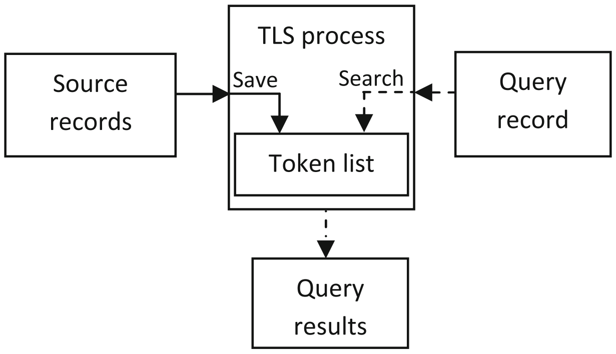


Fig. 4 The process of TLS

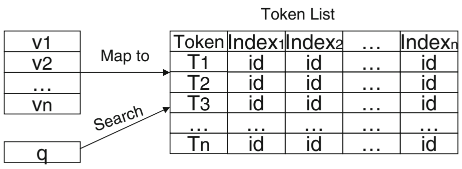


Fig. 5 An example of the TLS process

employ such a vector model to transform records to the same dimensional identifiers. TLS avoids the problem of identifier sparsity and the curse of dimensionality by only considering the tokens contained in the records. Typically, TLS reads tokens in a record one by one, and checks if the token is in the token list. If not, TLS inserts the token into the token list and stores the index of the record in this token's entry in the token list. Otherwise, TLS only stores the index of the record in the entry of this token. After reading all the records in the database, a final token list is constructed. The procedure of building the token list is shown in Algorithm 1.

Algorithm 1 Procedure for constructing the token list.

```
 1: for each record in a source record do
 2:   for each token in a record do
 3:     Upper case the token
 4:     if the token is in current token list then
 5:       Save the index of the record to the token's entry
 6:     else
 7:       Insert the token into the token list and also save the index of the record
 8:       Sort the token list lexicographically
 9:     end if
10:   end for
11: end for
```

The following example demonstrates how TLS constructs the token list. Assume that the records in a database are as follows:

id_1: Ann | Johnson | 16 | Female | 248 | Dickson | Street
id_2: Ann | Johnson | 20 | Female | 168 | Garland
id_3: Mike | Smith | 16 | Male | 1301 | Hwy
id_4: John | White | 24 | Male | Fayetteville | 72701

After reading the records id_1, id_2, id_3, and id_4, TLS builds a token list table as shown in Fig. 6. As a result, all records are grouped by their tokens: the records whose indices are linked with the same token have this token in common. Therefore, those records can be considered similar records to a certain degree.

3.1.2 *Searching the token list*

When searching for records similar to a query record, TLS reads each token of the query record and locates this token in the token list. If TLS finds the token in the token list, it returns all the indexed records in this token's entry. Otherwise, it means that this token is

Fig. 6 An example of a constructed token list

Token	Index	Index
16	1	3
20	2	
24	4	
168	2	
248	1	
1301	3	
72701	4	
ANN	1	2
DICKSON	1	
FAYETTEVILLE	4	
FEMALE	1	2
GARLAND	2	
HWY	3	
JOHN	4	
JOHNSON	1	2
MALE	3	4
MIKE	3	
SMITH	3	
STREET	1	
WHITE	4	

not in the token list. TLS then repeats this process with the next token in the query record. Algorithm 2 shows the procedure for locating records similar to a query record.

For example, given a query record:

q: Ann | Johnson | 20 | Female | 168 | Garland

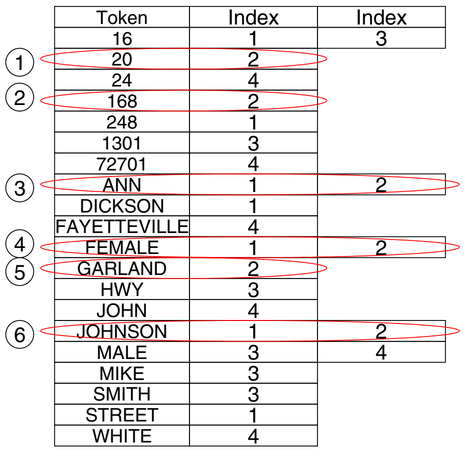
Because the query record q has tokens “20”, “168”, “ANN”, “FEMALE”, “GARLAND”, and “JOHNSON”, TLS starts checking the collections linked with the “20”, “168”, “ANN”, “FEMALE”, “GARLAND”, and “JOHNSON” tokens. Figure 7 shows that the $collection_1(C_1)$, C_2, C_3, C_4, C_5 and C_6 are the groups of records which have one token in record q. Then, TLS unions the collections:

$$Similar\ records = C_1 \cup C_2 \cup C_3 \cup C_4 \cup C_5 \cup C_6$$

The records id_1 and id_2 are returned since their indices are linked with the tokens of q in the token list. Therefore, id_1 and id_2 are similar to q.

Algorithm 2 Procedure for finding similar records of query records.

```
1: for each query record do
2:   for each token in the query record Do
3:     Upper case the token
4:     if the token is in current token list then
5:       Return all the indices linked with the token
6:     end if
7:     Merge all the returned indices and remove the duplicates
8:   end for
9: end for
```



	Token	Index	Index
	16	1	3
1	20	2	
	24	4	
2	168	2	
	248	1	
	1301	3	
	72701	4	
3	ANN	1	2
	DICKSON	1	
	FAYETTEVILLE	4	
4	FEMALE	1	2
5	GARLAND	2	
	HWY	3	
	JOHN	4	
6	JOHNSON	1	2
	MALE	3	4
	MIKE	3	
	SMITH	3	
	STREET	1	
	WHITE	4	

Fig. 7 An example of searching the token list

3.1.3 Inserting and deleting

In addition to searching, insertion and deletion are two important operations in a database. Insertion stores a new record into the database; deletion deletes a record from the database. When saving a new record into the database, the new record's tokens are scanned. If a scanned token of the new record has already been stored in the token list, the index of the new record is stored in the token's entry in the token list. Otherwise, the scanned new token is stored into the token list, and the index of the new record is stored in the new token's entry.

When an old record needs to be deleted from the database, the token list is searched for all the tokens of the target record. When a token is located, the index of the target record is deleted from the token's entry. If the index of the target record is the only one that is in the located token's entry, both the located token and its index are deleted from the token list. Finally, the target record is deleted from the database.

3.1.4 Performance analysis of TLS

In TLS, there are two main memory requirements. One is the memory for storing the source records, and the other is the memory for storing the token list. The consumption of the memory for storing the token list is related to the number of the tokens contained in each source record, and the number of source records and unique tokens in the database. If the average number of tokens contained in a source record is large, the memory requirement of the token list is high because the index of a record needs to be stored with every token of the record. With an increase of the source records, the memory requirement of the token list also increases.

The time consumption of TLS is for constructing the token list and searching for similar records. When constructing the token list, TLS needs to go through every token of every source record. If there are n source records and the average number of tokens in a source record is m, the time complexity of constructing the token list is $O(nm)$. When querying a record, if the number of tokens in the query record is m_q, TLS uses $O(m_q)$ to locate all the index collections of the records that include the tokens of the query record.

For filtering out records that are insufficiently similar to the query record, after collecting all the similar records of a query, similarity can be calculated between the located similar records and the query record:

$$similarity = \frac{|A \cap B|}{|A \cup B|/2}. \tag{1}$$

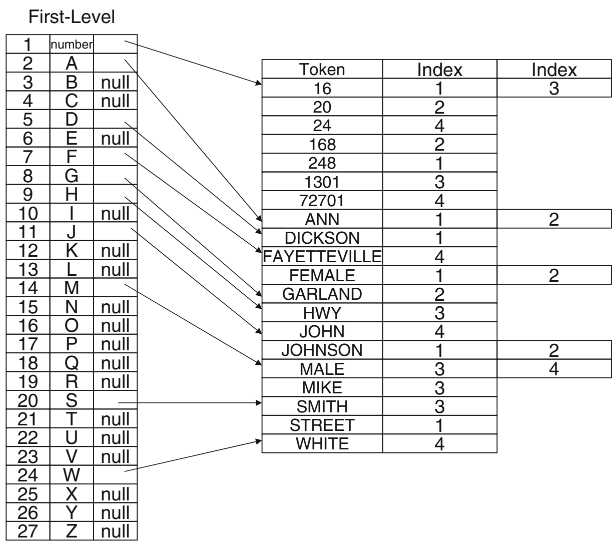
The user can set the threshold of similarity and export only the records that have at least a certain similarity with the query record. For example,

A: $Ann|Johnson|16|Female$
B: $Ann|Johnson|20|Female$

The similarity of record B to record A is 3/4 = 0.75.

3.2 Optimized token list based searching scheme

Because of the tremendous number of tokens in a massive database, the length of the token list is very long, potentially leading to long delays for locating tokens in the token list. We then use the index-tree and hash table structures to reduce search time. Tokens in the token list are sorted lexicographically, so we can use a small index table to index each group of tokens with the same first letter in the token list. Figure 8 shows an example of an index-tree structured token list that consists of ten numbers and twenty six letters. In the first level of the index-tree, each pointer points at the group of tokens whose first letter is the same as the first level index letter. When receiving a query, TLS first checks the first letter of the query record's token. For example, given a query token "ANN" with first letter "A", the Optimized Token list based Search schemes (OTS) first checks the top level index to locate "A". Next, it uses the pointer of "A" to locate the group of all tokens which have "A" as the first letter. Then, OTS starts searching the token list table to find the token "ANN". Lastly, OTS retrieves all the records (e.g., id_1 and id_2) whose indices are in the entry of



1	number	
2	A	
3	B	null
4	C	null
5	D	
6	E	null
7	F	
8	G	
9	H	
10	I	null
11	J	
12	K	null
13	L	null
14	M	
15	N	null
16	O	null
17	P	null
18	Q	null
19	R	null
20	S	
21	T	null
22	U	null
23	V	null
24	W	
25	X	null
26	Y	null
27	Z	null

Token	Index	Index
16	1	3
20	2	
24	4	
168	2	
248	1	
1301	3	
72701	4	
ANN	1	2
DICKSON	1	
FAYETTEVILLE	4	
FEMALE	1	2
GARLAND	2	
HWY	3	
JOHN	4	
JOHNSON	1	2
MALE	3	4
MIKE	3	
SMITH	3	
STREET	1	
WHITE	4	

Fig. 8 An example of index-tree structured token list

“ANN” in the token list. With growth in the number of records in the database, more levels of index-trees can be generated based on the letters in different positions in a token.

Alternatively, hash tables can be used for indexing the tokens in the token list. We can directly hash each unique token and replace the tokens in the token list with the hash values. Figure 9 shows an example of a hash table structured token list. For example, if a token of a query record is “ANN”, the OTS scheme hashes “ANN” to get the hash value 1, then directly locates the position of hash value 1 in the hash table structured token list to collect all the records whose indices are in the entry of this hash value. In Fig. 9, the records id_1 and id_2 are returned as the records similar to the query record, since they all have the token “ANN”.

After collecting the similar records of a query, the refinement process is conducted. Similarity is calculated between the returned similar records and the query record using similarity measure. The user can set the threshold which is the range of similarity to control the desired similar records. The returned records with similarity higher than the threshold are exported as the final querying results.

3.3 Multi-token list based searching scheme (MTLS)

In the single-token based token list, for each token of a query, a large number of records will be retrieved. In these retrieved records, many of them share only one token with the query record that may not be interested by the requester user. Moreover, the further refinement on the retrieved results to get rid of the overlapping and unwanted records generate considerable overhead. To handle this problem, we introduce the Multi-token List based Searching scheme (MTLS).

3.3.1 Constructing token list

Definition 2 A multi-token or n-token refers to a token composite consisting of n keywords. For example, ‘Ann Johnson” is a 2-token based token in the previous example.

Fig. 9 An example of hash table structured token list

ID	Token	Index	Index
1	ANN	1	2
2	MIKE	3	
3	JOHN	4	
4	JOHNSON	1	2
5	SMITH	3	
6	WHITE	4	
7	16	1	3
8	20	2	1
9	24	4	
10	FEMALE	1	2
11	MALE	3	4
12	ANN248	1	
13	168	2	
14	1301	3	
15	DICKSON	1	
16	STREET	1	
17	GARLAND	2	
18	HWY	3	
19	FAYETTEVILLE	4	
20	72701	4	

MTLS builds a token list, in which each entry contains a multi-token (i.e., n-token ($n > 1$)). In MTLS, all the source records are parsed into single-tokens. These single-tokens are concatenated into n-tokens. That is, a multi-token is formed by combining n keywords of each record in a permutation way to retrieve all the possible combination of the keywords in each record. Figure 6 with Fig. 10 show examples for a single-token based token list and a two-token based token list, respectively. The token list contains each n-token and the indices of records that contain the n-token. For example, Fig. 10 shows that record1 and record2 contain the 2-token "Ann Female".

Algorithm 3 shows the pseudo-code of multi-token based token list construction. To build a token list for a large amount of records, MTLS reads a record $v[i]$ at one time and parses it into single-tokens. If $n > 1$, MTLS concatenates every n tokens. MTLS inserts the newly created n-token along with the index of the record to the token list if the n-token does not exist in the token list. Otherwise, MTLS only needs to store the index of the record to the entry of the n-token. Then, MTLS reads the next record $v[i + 1]$ and repeats the same process.

We show an example to explain the MTLS token list construction process. Assume that the records in a database are as follows: the index of record id_1 is 1; the index of record id_2 is 2; the index of record id_3 is 3; and the index of record id_4 is 4.

id_1:	Ann | Johnson | 16 | Female | 248 | Dickson | Street
id_2:	Ann | Johnson | 20 | Female | 168 | Garland
id_3:	Mike | Smith | 16 | Male | 1301 | Hwy
id_4:	John | White | 24 | Male | Fayetteville | 72701

To construct a single-token based token list, MTLS parses each record into several single-tokens, and inserts these single-tokens into a token list. For a number of identical single tokens, only one of them is used in the token list, and the indices of records having the token are associated with the token in the token list. Figure 6 shows an example of single-token based token list.

To build a two-token based token list, for each record, MTLS parses it into two-tokens. For example, the single-tokens in record 1 is "Ann", Johnson", "16" , "Female", "248", "Dickson", "Street". Then, MTLS concatenates every two of the tokens and inserts it into the token list, that is, "Ann Johnson", "Ann 16", "Ann Female", "Ann 248", "Ann Dickson", "Ann Street", "Johnson 16", "Johnson Female" and so on. Figure 10 shows an example of part of the two-token based token list about record 1 and record 2.

Fig. 10 An example of a two-token based token list

ID	n-token	Index	Index
1	ANN JOHNSON	1	2
2	ANN 16	1	
3	ANN FEMALE	1	2
4	ANN 248	1	
5	ANN DICKSON	1	
6	ANN STREET	1	
7	JOHNSON 16	1	
8	JOHNSON FEMALE	1	2
9	JOHNSON 248	1	
10	JOHNSON DICKSON	1	
11	JOHNSON STREET	1	
...	......	...	...

3.3.2 Performance analysis of MTLS

Comparing Fig. 6 with Fig. 10, we can find that although Fig. 10 only depicts the two-tokens of two records, its list size is already much larger than the single-token based list with four records. This is because for a record with m keywords, the number of n-token is: $C_m^n = m \times (m-1) \times ...(m-n-1)/n!$. With the increase of n, the permutation time to calculate the multi-token will increase. Therefore, it is important to determine a suitable value for n.

The n-token based token list could be very long. In order to efficiently retrieve data, each n-token is hashed to an integer and stored in a hash table for efficient n-token search and storage. Then, the data structure of the token list is <hashValue, record indices>. Thus, for every n-token, MTLS can locate it in the token list directly with time complexity of O(1).

Algorithm 3 Pseudo-code for multi-token based list construction

```
 1: tokenlist $T = null$;
 2: for each $v[i]$ of a record do
 3:    $v[i]$ is parsed into tokens;
 4:    Concatenate every $n$ tokens into $t[j]$
 5:    if $t[j]$ exists in $T$ then
 6:       Store the index of the record to the entry of $t[j]$
 7:    else
 8:       Add $t[j]$ into $T$
 9:       Store the index of the record to the entry of $t[j]$
10:    end if
11: end for
12: return $T$;
```

3.3.3 Searching token list

The querying process is similar to the token list construction process. The multi-tokens serve as new tokens for the similarity searching. In the searching process, the query record is also parsed to single-tokens. Then, every n of these token are concatenated together with different order to form a n-token. For every n-token, MTLS hashes it into an integer. Each n-token is identified in the token list using the hash value, and corresponding indices are fetched. Then, the intersection of the fetched indices are the indices of the located records for the query. Finally, MTLS uses the refinement on the records of the identified indices for the query results. The multi-token based data searching reduces the number of retrieved records by increasing the similarity limitation to n $(n > 1)$ keywords. It reduces the refinement time and hence decreases query latency.

3.4 Applications of token list based searching schemes

Our proposed token list based information search schemes provide similar information search service by exploring the similarities between records and a given query. It works in a multi-dimensional massive database. Such a massive database consisted of a large number of data records, and each record is described by a number of tokens. Our schemes offer high searching accuracy and low consumption in memory and latency in a multi-dimensional massive database. For example, in a bank customer dataset, these schemes can find customers based on a query consisting of a number of tokens. In a full-text databases (e.g., web pages), each data record is represented in plain text. To apply our schemes to full-text databases, we can transform each plain-text data file to a data record consisting of a number of tokens by executing a pre-processing step. For example, each web page has keywords including phrases found throughout the page content and title tag (Alimohammadi 2003; Qi and Davison 2009). Also, some popular information retrieval techniques such as *tf-idf* theme, vector space model and latent semantic indexing (Salton and McGill 1983; Berry et al. 1999; Nejdl et al. 2003; Halevy et al. 2003; Aberer et al. 2003; Nejdl et al. 2003) can extract keywords from full-text files. Then, the token list to create the multi-dimensional massive database can be built. However, these pre-processing techniques are orthogonal to our study in this paper. The details of such techniques are beyond the scope of this paper. After the plain-text data records are transformed to data records, our schemes can be directly applied to the transformed database for similar information search. Another type of databases include token types as well as spatial-temporal dimensions such as the sensed data (e.g., Car, 8:00am, (Latitude, Longitude)=(40, 34)) in a wireless sensor network. In this case, we can directly regard the spatial dimension and the temporal dimension as individual dimensions in the multi-dimensional space, and then can use our schemes for similar information search on the databases directly.

4 Performance evaluation

We implemented the proposed schemes and conducted experiments on these schemes, using E2LSH 0.1 (Andoni 2005) as an additional comparison. E2LSH 0.1 is a simulator for high-dimensional near neighbor search based on LSH in Euclidean space developed by MIT. Our testing dataset is a set of synthetically generated names and addresses that imitate the properties of actual customer data in a simulated CDI database offered by the Acxiom Corporation, which is a major provider of integrated customer information globally. There are 10,000 source records and 20,591 unique tokens total in the database. Therefore, the dimension of the space and the length of every record's binary identifier is 20,591. We randomly selected 97 query records from the source records unless otherwise specified. Each record has a different number of tokens, resulting in an average number of tokens in a record equals 10. In E2LSH 0.1, in the hash function $h_{a,b}(v) = \left\lfloor \frac{a \cdot v + b}{w} \right\rfloor$, w was set to 4 as an optimized value (Andoni and Indyk 2005). The Euclidean space distance threshold of R between identified similar records and the query in the LSH-based scheme was set to 3 in all experiments.

4.1 Token list based information search scheme

In these experiments, we use "*Token list*" to denote the OTS with the hash table structured token list.

4.1.1 Memory cost

Figure 11 plots the total memory requirements of LSH and *Token list*. From Fig. 11, we can see that LSH incurs much higher memory consumption than *Token list*. LSH can only process the identifiers of records that have the same dimension (i.e. 20,591), but *Token*

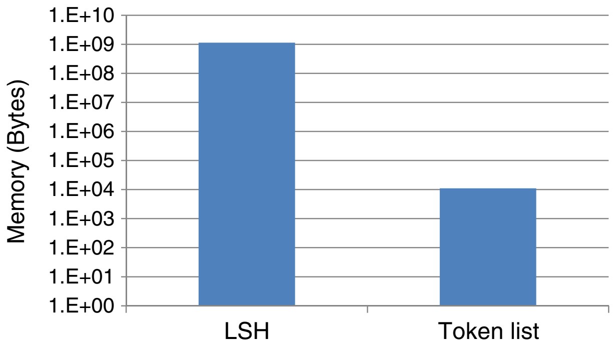


Fig. 11 Total memory usage of LSH versus *Token list*

list does not have that requirement. In addition, in *Token list*, there is only one token list table. Therefore, LSH's memory consumption is significantly higher than that of *Token list*. This experimental result confirms the effectiveness of *Token list* in reducing the memory consumption of LSH.

4.1.2 Time cost

Figure 12 shows the query latency of the linear searching scheme, the kd-tree searching scheme, LSH, and *Token list* versus the number of query records. We can see that the query time of the linear searching method is the highest, and LSH and *Token list* lead to faster similar records location than the kd-tree method. Given n pieces of records in a database, traditional methods based on tree structures need $O(\log n)$ time for a query, and the linear searching method needs $O(n)$ time for a query. LSH and *Token list* need $O(T)$ time to locate all the similar records, where T is a constant. The results confirm that LSH and *Token list* achieve much faster search speeds than other proximity search methods. Because the kd-tree method produces relatively much higher time cost than LSH and *Token list*, below we only compare LSH and *Token list*.

Figure 13 displays the total query time of LSH and *Token list*. We can see that LSH leads to dramatically longer search times than *Token list*. This is because *Token list* only needs to calculate one hash value for each token of each record, but LSH needs to generate over 2,000 hash values for each record. Furthermore, the refinement step in LSH exacerbates the search

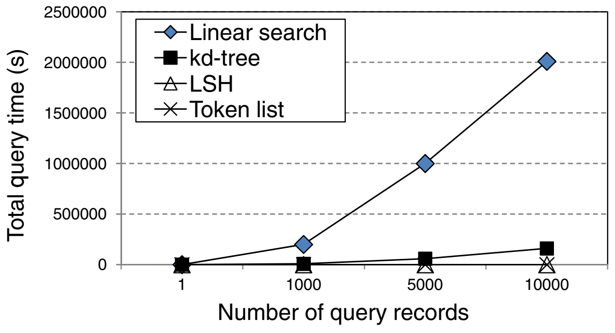


Fig. 12 Total query times of linear searching, kd-tree searching, LSH, and *Token list*

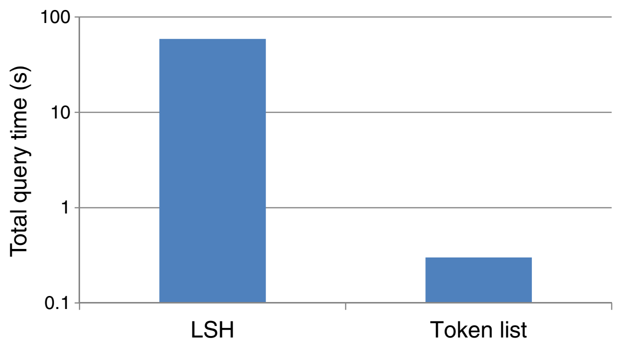


Fig. 13 Total query time of LSH versus *Token list*

latency. This is confirmed by Fig. 14, which shows the number of returned results of *Token list* and LSH before and after the refinement step. We can observe that before the refinement phase, LSH returns many more records than after refinement. In the refinement phase, LSH needs to calculate the Euclidean space distance, which contains addition, subtraction, and squaring operations to filter the records that are outside of a predefined range. Therefore, the total query time of LSH is much longer than the total query time of *Token list*. In this experiment, if one token contained in the query record is also contained in a source record, we consider the source record is a similar record to the query record. Thus, there is no refinement step in *Token list*.

4.1.3 Effectiveness

False positive results are those records that are identified as similar records but actually are not similar to the query record. An effective method should return fewer false positive records. In this experiment, if one token contained in query record is also contained in a source record, we consider the source record as similar to the query record. Because *Token list* uses the tokens of query records to query the token list, it ensures that the returned records are all similar to the query. However, in LSH, due to the identifier sparsity of records, Euclidean space distance computation may return some records that do not have a common token with the query. Figure 15 shows the percentage of similar records in the

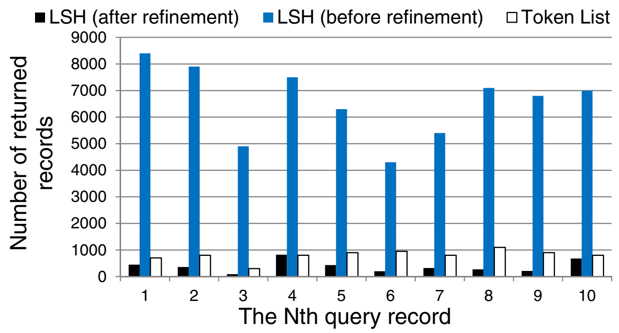


Fig. 14 Number of returned records by LSH and *Token list*

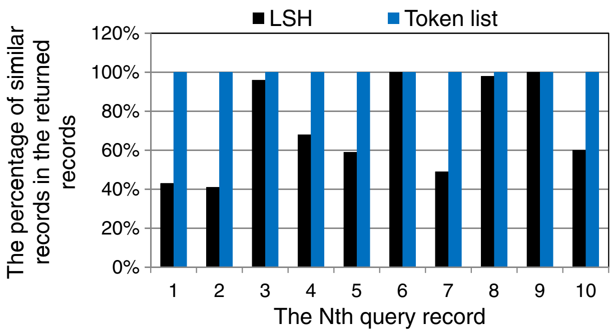


Fig. 15 The percentage of similar records in returned records

returned records of LSH and *Token list*. The similar records have at least one token in common with the query record. We can see that in most cases, more than 50 % of returned records are false positives in LSH. In contrast, *Token list* has 0 false positives. The experimental results imply that *Token list* has higher effectiveness than LSH in reducing false positives in searching.

4.1.4 Accuracy

An important metric for a searching scheme is accuracy, which is used to measure how many actual similar records a scheme will miss in searching. The definition of accuracy is as follows:

$$Accuracy = \frac{Number\ of\ located\ similar\ records}{Number\ of\ similar\ records}. \tag{2}$$

A scheme with higher accuracy misses fewer similar records. Figure 16 shows what percentage of existing similar records can be returned for each query (i.e., accuracy rate). *Token list* can return 100 % of similar records for each query. However, LSH can return a high of 65 % and a low of 4 % for similar records. Compared with LSH, *Token list* has a much higher accuracy rate.

Another experiment is conducted on LSH and *Token list*. We randomly chose one record from the 10,000 records, changed one token to generate a new record as the query record every time, and then used LSH and *Token list* to conduct the query in the 10,000 source

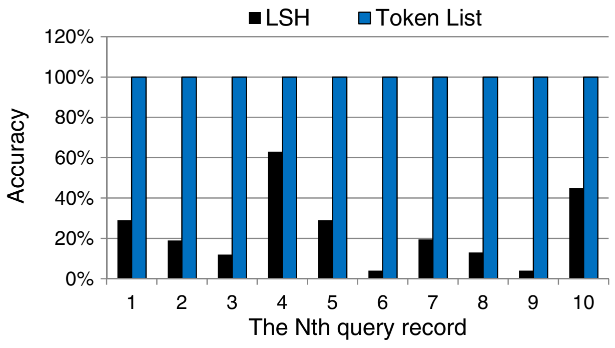


Fig. 16 The percentage of similar records that are returned

records to test if the methods could find the original record. Table 1 shows the similarity between the query record and the original record, and the query result. If the original record can be found, it is marked “Y”; otherwise, it is marked as “N”. From the table, we see that *Token list* can find all the original records even if their similarity is as low as 0.1, but LSH only can find the records that have similarity no less than 0.4. Therefore, *Token list* has higher accuracy than LSH.

From the above experiments, we obtain the following conclusions:

(1) LSH and *Token list* improve the search speed of linear search and tree structure search.
(2) *Token list* outperforms LSH in terms of efficiency and effectiveness. Compared to LSH, *Token list* has higher accuracy, faster query speeds, and lower memory requirements.
(3) LSH requires all the input data records to have the same dimension, but *Token list* can work on different dimensional datasets. This is the reason that *Token list* returns fewer false positive records and false negative records than LSH in a massive database with a large number of tokens.

4.2 Optimized token list based search schemes

4.2.1 Memory cost

We then test the performance of TLS, *TLS with index table*, and *TLS with hash table*. The total number of source records was 1,000, each source record has different unique tokens. There are 10 unique tokens for each record. We randomly chose 100 records from the source records as queries. Figure 17 shows the memory consumption of LSH, TLS, *TLS with index table*, and *TLS with hash table*. From the figure, we can see that the memory consumptions of TLS, *TLS with index table*, and *TLS with hash table* are much less than that of LSH. Because the tokens of source records are unique tokens, all the tokens are saved in the token list table. The index of each source record is also saved with the tokens in the source records. In LSH, each record is transformed to m IDs and LSH has m hash tables to store record indices. In this experiment, LSH uses 91 hash tables ($m = 91$) to achieve fast query speeds, and the dimension of record identifiers is 10,000. Therefore, LSH needs more memory than TLS, *TLS with index table*, and *TLS with hash table*. We also see that *TLS with index table* and *TLS with hash table* consume almost the same amount of memory as TLS. *TLS with index table* only needs to maintain an additional index table for an alphabet list, which takes

Table 1 Accuracy test

Similarity	LSH	***Token list***
1.0	Y	Y
0.9	Y	Y
0.8	Y	Y
0.7	Y	Y
0.6	Y	Y
0.5	Y	Y
0.4	Y	Y
0.3	N	Y
0.2	N	Y
0.1	N	Y

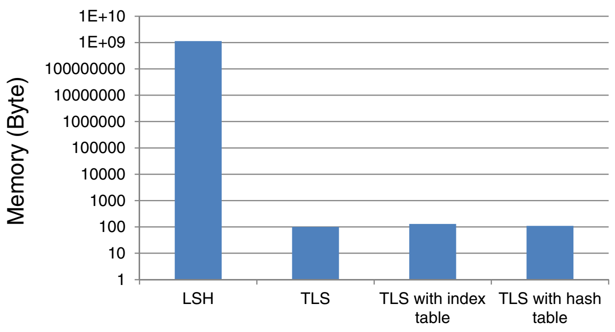


Fig. 17 Memory consumption

a very small amount of memory, to point to different parts of the token list. Thus, it does not consume much additional memory. *TLS with hash table* maintains the hash values of tokens rather than tokens in the token list. As a result, it reduces the memory consumption of TLS.

4.2.2 Time cost

Figure 18 displays the total query time of LSH, TLS, *TLS with index table* and *TLS with hash table*. From the figure, we can notice that the total query time of TLS is much shorter than that of LSH. There are two main steps that take most of the query time of LSH: computing hash values for the queries and distance computation between queries and source records for refinement. In TLS, *TLS with index table*, and *TLS with hash table*, the total query time contains the time for deriving all the tokens from the queries and the time for searching the token list table. The number of tokens in a record is only 10, much less than the dimension of the identifiers in LSH (10,000), so the query speed of TLS, *TLS with index table*, and *TLS with hash table* is faster than that of LSH. As *TLS with index table* and *TLS with hash table* only search a portion of the token list, their total query time periods are much shorter than that of TLS. Based on the hash ID of the query token, *TLS with hash table* can directly find the query token in the token list. *TLS with index table* needs to search the index of a alphabet list, and then search the token list. As a result, *TLS with hash table* has a faster query speed than *TLS with index table*.

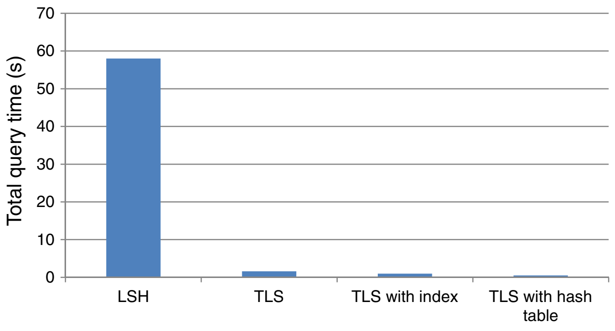


Fig. 18 Total query time of LSH and TLS

4.2.3 Effectiveness

Because TLS, *TLS with index table*, and *TLS with hash table* use the tokens of query records to query the token list, they can ensure that the returned records are all actual similar records of the query. LSH transforms data records into binary representations, and calculates the Euclidean space distance between the located records and the query to identify similar records. Thus, it may return some records that do not have a common token with the query. Figure 19 shows the percentage of actual similar records in the identified similar record list of LSH, TLS, *TLS with index table* and *TLS with hash table* after the refinement phase. We can see that in most of the cases, more than 50 % of returned records are false positives in LSH. In contrast, TLS, *TLS with index table*, and *TLS with hash table* do not have any false positives. The experimental results confirm that TLS has higher effectiveness than LSH in reducing false positives in searching, and *TLS with index table* and *TLS with hash table* achieve similar performance to TLS in reducing false positives.

4.2.4 Accuracy

A scheme with higher accuracy misses fewer similar records. Figure 20 shows the accuracy rate of each query in different schemes. Because TLS, *TLS with index table*, and *TLS with hash table* only return the source records with the same token as a query, they can return 100 % of the similar records for each query. However, there are false positives in the similar record list of LSH. Thus, the accuracy of LSH cannot be 100 %. In contrast, LSH returns a high of 65 % and a low of 4 % of the total similar records. Therefore, compared to LSH, TLS, *TLS with index table*, and *TLS with hash table* have much higher accuracy.

We use OTS to represent both *TLS with index table* and *TLS with hash table*. From the above experiments, we can conclude that OTS greatly reduces the query time of TLS without increasing the memory consumption of TLS or reducing its accuracy and effectiveness in similarity search.

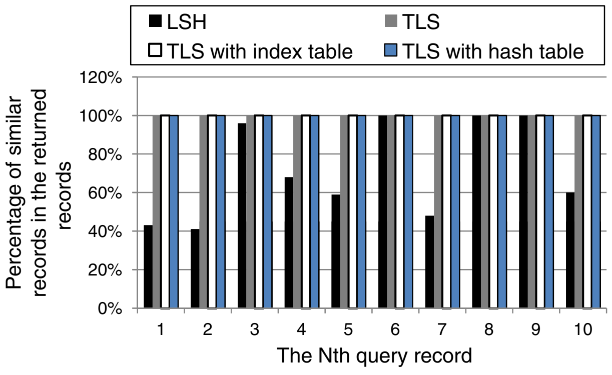


Fig. 19 The percentage of similar records in returned records

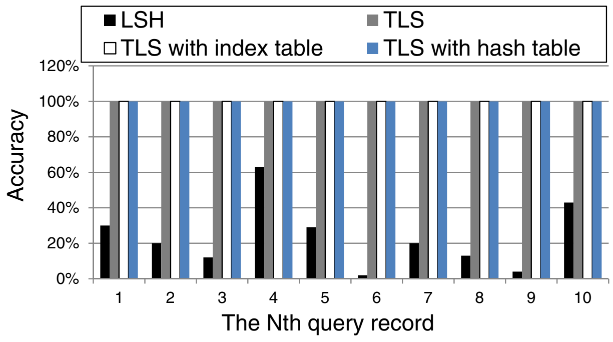


Fig. 20 Accuracy of each query

4.3 Multi-token list search scheme (MTLS)

4.3.1 Time cost

We use MTLS-n to represent n-token list in MTLS. Figure 21 shows the query time of different MTLS methods, measured by the time period from the time when a query is initiated to the time when the requester receives the response (without the filtering process). From the figure, we can see that MTLS-1 generates the longest query time, and MTLS-2 produces the fastest query speed than other methods. Query time consists of the time for sorting tokens of query records and the time for searching the token list. The former is the time for getting all the single-tokens of query records, and generating the combinations of the query tokens. The latter is the time for retrieving the similar records of the query from the token list based on different combinations of the query tokens.

From Fig. 21, we notice that sorting query tokens takes much less time than searching the token list in all the methods. Because each query record only contains about 10 tokens, it is easy to generate the combination of the query tokens. However, when searching token list according to the combination of query tokens, MTLS needs to find the location of all the records containing the combination. Because there are more records that have one common token with query record than the records that have two or three common tokens with

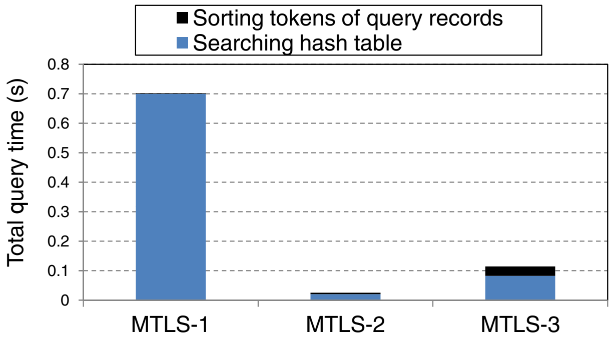


Fig. 21 Total query time for MTLS

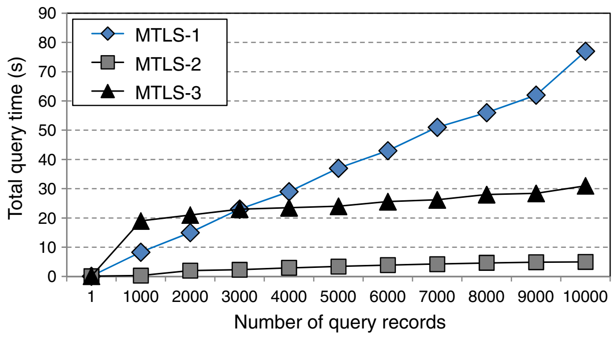


Fig. 22 Total query time with different numbers of query records for MTLS

query record, searching token list time of MTLS-1 is longer than other methods. Comparing MTLS-2 with MTLS-3, we found that MTLS-2 has less searching token list time than MTLS-3. If a record has 10 tokens, there are 90 combinations for two tokens and 720 combinations for three tokens. Therefore, MTLS-2 has to look up the token list for 90 times, while MTLS-3 has to look up the token list for 720 times that is eight times more than MTLS-2, which leads to long searching token list time. With the increase of the number of token combinations, the time for sorting query tokens increases. Therefore, MTLS-2 can achieve short query time and it is more efficient than MTLS-1 and MTLS-3.

We also conducted experiment with different number of query records as shown in Fig. 22. The result of Fig. 22 confirms that MTLS-2 achieves the fastest query speed in all three methods. We also see that as the number of queries increases, the total query time increases. The query time increasing rates of MTLS-2 and MTLS-3 are slower than that of MTLS-1. When the number of query records is less than 3,000, the query time of MTLS-3 is even longer than that MTLS-1. Therefore, MTLS-3 only works well with a large number of query records.

Figure 23 presents the total processing time of MTLS-1, MTLS-2 and MTLS-3. Total processing time includes of the time for sorting token of source records and query time. From Fig. 23, we can see that sorting tokens of source records takes most of the total processing time. With the increase of the number of token combinations, the time for sorting tokens of source records increases. MTLS-3 has the longest sorting token time. If a record

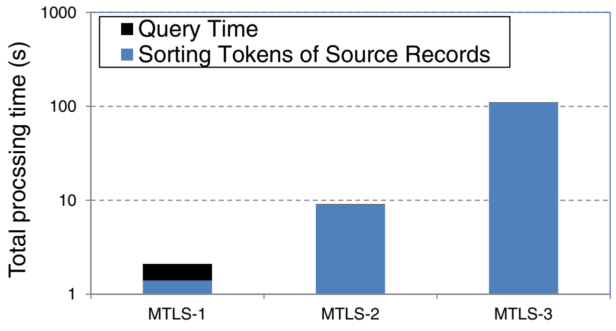


Fig. 23 Total processing time for MTLS

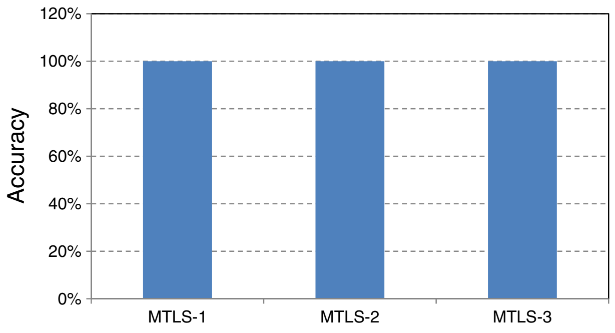


Fig. 24 Percentage of true positives for MTLS

has ten tokens, MTLS-1 can make 10 token combinations for the record; MTLS-2 can generate 90 combinations; MTLS-3 produces 720 combinations which is the most combinations in the these methods. Therefore, the sorting token time of MTLS-3 is dramatically higher than other methods. The long tokens of source records leads to long total processing time.

4.3.2 Accuracy

Figure 24 shows the percentage of true positives in the located records. We can observe that all the records located by MTLS are similar as queries. MTLS generates all the combinations for the tokens of source records, and it assigns a unique hash code for each token combination. When searching a query, MTLS looks up the hash table base on the query token combination. Therefore, MTLS only return the records have the same token combinations with query.

4.3.3 Memory cost

Figure 25 plots a comparison of memory costs between LSH and different MTLS schemes. In LSH, the memory cost includes storage of source record IDs and hash tables; while in MTLS, the memory cost includes storage of token list and associated source record IDs. In LSH, every resource is transformed into a vector of 0/1 binary number, and the length of this vector equals to the number of unique tokens in the dataset, which is 20,591 in the

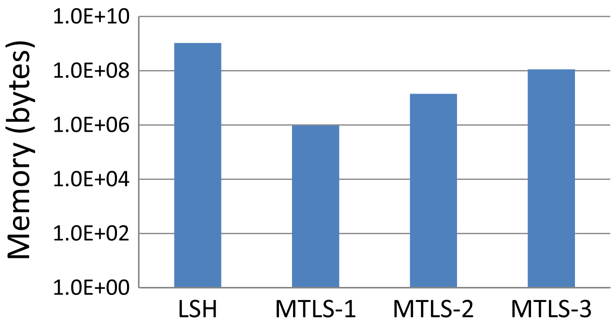


Fig. 25 Memory cost for MTLS

experiments. Also, LSH uses a number of hash tables to store the hash values of each record. LSH requires more memory than MTLS. In MTLS-1, the identifier of a record is stored in the token list. As the average number of tokens in a record equals 10, so the average number of generated IDs stored in the token list of each record equals 10. Also, MTLS needs to maintain a token list, whose size is based on the number of tokens we use. The more tokens we use, the larger number of different combinations of tokens is generated by every record, so the memory cost follows MTLS-3>MTLS-2>MTLS-1.

5 Conclusions

Proximity search in multi-dimensional massive databases is of great importance for many applications in our daily life. A locality sensitive hashing (LSH) scheme can improve the search efficiency of linear searching and tree structure based searching methods. However, LSH needs a large memory space for storing the transformed source records and hash tables, and its Euclidean space distance calculation leads to long query times and low accuracy. To deal with these problems, this paper has studied a token list based proximity searching scheme (TLS) that can successfully and efficiently search proximity information in a massive database. TLS builds a token list containing all of the unique tokens in the records. It groups records with the same token together. A query is only mapped to the groups which have the query's tokens.

To reduce the search latency of TLS, this paper further has studied optimized TLS based on index-tree and hash table structures (OTS). An index-tree structure orders the tokens in the token list and constructs an index table based on the tokens. Searching the token list can be started from the entry of the token list supplied by the index table. A hash table structure assigns a hash ID to each token so that the query token can be directly located in the token list according to the hash ID.

The single-token based TLS may generate high overhead during result refinements. Also, it is unlikely that the returned records sharing only one token with the query record are of interest to the users. The paper then further investigates how token size affects query time and query accuracy in proximity search by building token lists with different multi-token sizes.

Experimental results show the high performance of TLS compared with LSH, linear, and tree-structured searching. TLS achieves higher efficiency and effectiveness than LSH in terms of memory and time consumption, and searching accuracy. Experimental results also show that OTS greatly reduces the query time of TLS without increasing the memory consumption of TLS or reducing its accuracy and effectiveness in similarity search. Furthermore, the results show that two-token based token list proximity search can obtain the best query performance in terms of query accuracy and query time using our test dataset. In the future, we will use other datasets to verity the high performance of two-token based token lists.

Acknowledgments This research was supported in part by U.S. NSF grants IIS-1354123, CNS-1254006, CNS-1249603, OCI-1064230, CNS-1049947, CNS-0917056 and, CNS-1025652, Microsoft Research Faculty Fellowship 8300751, Microsoft Research Faculty Fellowship 8300751, and the United States Department of Defense 238866. Early versions of this work were presented in the Proceedings of DMIN'08 (Li et al. 2008) and ICCIT'08 (Shen et al. 2008). We would like to thank Mr. Yuhua Lin for his valuable comments in addressing the review feedback.

References

Aberer, K., Cudrè-Mauroux, P., Hauswirth, M. (2003). The chatty web: emergent semantics through gossiping. In *Proceedings of the 12nd international world wide web conference*.

Alimohammadi, D. (2003). Meta-tag: a means to control the process of web indexing. *Online Information Review*, *27*(4), 238–242.

Andoni, A. (2005). Lsh algorithm and implementation (e2lsh). http://web.mit.edu/andoni/www/LSH/index.html.

Andoni, A., & Indyk, P. (2005). E2lsh 0.1 user manual. http://web.mit.edu/andoni/www/LSH/index.html.

Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A. (1994). An optimal algorithm for approximate nearest neighbor searching. In *Proceedings 5th ACM-SIAM symposium discrete algorithms*.

Bayer, R., & McCreight, E. (1970). Organization and maintenance of large ordered indices. In *Proceedings of ACM-SIGFIDET workshop on data description and access* (pp. 107–141).

Beckmann, N., Kriegel, H., Schneider, R., Seeger, B. (1990). The r*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD international conference on management of data* (pp. 322–331).

Bennett, K.P., Fayyad, U., Geiger, D. (1999). Density-based indexing for approximate nearest-neighbor queries. In *Proceedings of KDD*.

Bentle, J.L., Friedman, J.H., Finkel, R.A. (1977). An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, *3*(3), 209–226.

Berchtold, S., Keim, D.A., Kriegel, H.-P. (1996). The x-tree: an index structure for high-dimensional data. In *Proceedings of the 22nd international conference on very large databases* (pp. 28–39).

Berrani, S.A., Amsaleg, L., Grosr, P. (2003). Approximate searches: k-neighbors + precision. In *Proceedings of CIKM*.

Berry, M.W., Drmac, Z., Jessup, E.R. (1999). Matrices vector spaces, and information retrieval. *SIAM Review*, *41*(2), 335–362.

Blachman, N. (2007). Google guide, making searching even easier. http://www.googleguide.com/google_works.html.

Bohm, C., Berchtold, S., Keim, D.A. (2001). Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, *33*(3), 322–373.

Brin, S. (1995). Near neighbor search in large metric space. In *Proceedings of the 21st international conference on VLDB*.

Chaudhuri, S., Church, K., Konig, A., Sui, L. (2007). Heavy-tailed distributions and multi-keyword queries. In *Proceedings of SIGIR*.

Chen, H., Jin, H., Wang, J., Chen, L., Liu, Y., Ni, L. (2008). Efficient multi-keyword search over p2p web. In *Proceedings of WWW* (pp. 989–998).

Chen, H., Yan, J., Jin, H., Liu, Y., Ni, L. (2010). TSS: efficient term set search in large peer-to-peer textual collections. *TC*, *59*(7), 969–980.

Comer, D. (1979). The ubiquitous B-tree. *Computing Surveys*, *11*(2), 121–138.

Cover, T., & Hart, P. (1967). Nearest neighbor pattern classification. *IEEE Transactions of Information Theory*, *IT-13*(1), 21–27.

Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S. (2003). Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of DIMACS workshop on streaming data analysis and mining*.

Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S. (2004). Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the 20th annual symposium on computational geometry (SCG)*.

Deerwester, S., Dumais, S.T., Landauer, T.K., Fumas, G.W., Harshman, R.A. (1990). Indexing by latent semantic analysis. *Journal of the Society for Information Science*, *41*(6), 391–407.

Fagin, R. (1998). Fuzzy queries in multimedia database systems. In *Proceedings ACM symposium on principles of database systems*.

Filho, R.F.S., Traina, A.J.M., Traina, J.C., Faloutsos, C. (2001). Similarity search without tears: the omni family of all-purpose access methods. In *Proceedings of ICDE*.

Fu, A., Chan, P.M.S., Cheung, Y.L., Moon, Y.S. (2000). Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *VLDB Journal*, *9*(2), 154–173.

Gionis, A., Indyk, P., Motwani, R. (1999). Similarity search in high dimensions via hashing. In *Proceedings of international conference on very large data bases (VLDB)* (pp. 518–529).

Grossman, D.A., & Frieder, O. (2004). *iFlow: information retrieval*. The Netherlands: Springer.

Guttman, A. (1984). R-trees: a dynamic index structure for spatial searching. In *Proceedings of the SIGMOD conference* (pp. 47–57).

Halevy, A.Y., Ives, Z.G., Mork, P., Tatarinov, I. (2003). Piazza: data management infrastructure for semantic web applications. In *Proceedings of the 12nd international world wide web conference*.

Hu, J.J., Tang, C.J., Peng, J., Li, C., Yuan, C.A., Chen, A.L. (2005). A clustering algorithm based absorbing nearest neighbors. In *6th International conference of WAIM*.

Indyk, P., & Motwani, R. (1998). Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the 30th annual ACM symposium on theory of computing*.

Kleinberg, J.M. (1997). Two algorithms for nearest-neighbor search in high dimensions. In *Proceedings of ACM symposium on theory of computing (STOC)*.

Kruskal, J.B., & Wish, M. (1978). *Multidimensional scaling*. Beverly Hills: SAGE publication.

Kulkami, S., & Orlandic, R. (2006). High-dimensional similarity search using data sensitive space partitioning. *Lecture Notes in Computer Science (LNCS)*, *4080*(2006), 738–750.

Lam, H., Perego, R., Quan, N., Silvestri, F. (2009). Entry pairing in inverted file. In *Proceedings of WISE* (Vol. 5802, pp. 511–522).

Li, C., Chang, E., Garcia-Molina, H., Wiederhold, G. (2002). Clustering for approximate similarity search in high-dimensional spaces. *IEEE Transactions of Knowledge and Data Engineering*, *14*(4), 792–808.

Li, T., Shen, H., Rosequist, A. (2008). Token list based data searching in a multi-dimensional massive database. In *Proceedings of The 4th international conference on data mining (DMIN)*.

Loccoz, N.M. (2005). *High-dimensional access methods for efficient similarity queries*. Technical Report TR-2005-05-05, Universite De GENEVE.

Long, X., & Suel, T. (2005). Three-level caching for efficient query processing in large Web search engines. In *Proceedings of WWW* (pp. 257–266).

Luu, T., Skobeltsyn, G., Klemm, F., Puh, M., Zarko, I., Rajman, M., Aberer, K. (2008). AlvisP2P: scalable peer-to-peer text retrieval in a structured P2P network. *PVLDB*, *1*(2), 1424–1427.

Nejdl, W., Siberski, W., Wolpers, M., Schmnitz, C. (2003). Routing and clustering in schema-based super peer networks. In *Proceedings of IPTPS*.

Nejdl, W., Wolpers, M., Siberski, W., Löser, A., Bruckhorst, I., Schlosser, M., Schmitz, C. (2003). Super-peer-based routing and clustering strategies for rdf-based peer-to-peer networks. In *Proceedings of the 12nd international world wide web conference*.

Niblack, C.W., Barber, R., Equitz, W., Flickner, M.D., Glasman, E.H., Petkovic, D., Yanker, P., Faloutsos, C., Taubin, G. (1993). The QBIC project: querying images by content using color, texture and shape. In *Proceedings of SPIE: storage and retrieval for image and video database*.

Panigrahy, R. (2006). *Nearest neighbor search using kd-trees*. Technical report, Stanford University.

Qi, X., & Davison, B. (2009). Web page classification: features and algorithms. *ACM Computing Surveys*, *41*(2), 1–31.

Salton, G., & McGill, M. (1983). *Introduction to modern information retrieval*. McGraw-Hill, International Student Edition.

Sellis, T., Roussopoulos, N., Faloutsos, C. (1997). Multidimensional access methods: trees have grown everywhere. In *Proceedings of the 23rd international conference on very large data bases*.

Shen, H., Li, Z., Li, T. (2008). An investigation on multi-token list based proximity search in multi-dimensional massive database. In *Proceedings of the international conference on convergence and hybrid information technology (ICCIT)*.

Skobeltsyn, G., Luu, T., Zarko, I., Rajman, M., Aberer, K. (2009). Query-driven indexing for scalable peer-to-peer text retrieval. *Future Generation Computing Systems*, *25*(1), 89–99.

Weth, C., & Datta, A. (2012). Multiterm keyword search in NoSQL systems. *IEEE Internet Computing*, *16*(1), 34–42.

White, D.A., & Jain, R. (1996). *Algorithm and strategies for similarity retrieval*. Technical Report VCL-96-101, University of California.

Yianlios, P.N. (1993). Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM symposium on discrete algorithms*.

Zolotarev, V.M. (1986). *One-dimensional stable distributions*. American Mathematical Society.