Bounded LSH for Similarity Search in Peer-to-Peer File Systems

Yu Hua[†]   Bin Xiao[‡]   Dan Feng[†]   Bo Yu[*]

[†]Huazhong University of Science and Technology   [‡]HongKong Polytechnic University   [*]Wayne State University
Wuhan, China   Kowloon, Hong Kong   Detroit, Michigan, USA
{csyhua,dfeng}@hust.edu.cn   csbxiao@comp.polyu.edu.hk   boyu@eng.wayne.edu

Abstract

Similarity search has been widely studied in peer-to-peer environments. In this paper, we propose the Bounded Locality Sensitive Hashing (Bounded LSH) method for similarity search in P2P file systems. Compared to the basic Locality Sensitive Hashing (LSH), Bounded LSH makes improvement on the space saving and quick query response in the similarity search, especially for high-dimensional data objects that exhibit non-uniform distribution property. We present simple and space-efficient Bounded-LSH to map non-uniform data space into load-balanced hash buckets that contain approximate number of objects. Load-balanced hash buckets in Bounded-LSH, in turn, require less number of hash tables while maintaining a high probability of returning the closest objects to requests. Our experiments based on synthetic and real-world datasets showed the feasibility, query and space efficiency of our proposed method.

1 Introduction

Peer-to-peer (P2P) file systems need to support similarity search over multi-dimensional data and maintain feature-rich objects, such as audio, images, videos, and other sensor data that exhibit high-dimensional features. Similarity search [1, 2] aims to find objects that have similar characteristics to the query object. When data objects are represented as d-dimensional feature vectors, the goal of similarity search for a given query object q is to find K objects that are closest to q according to distance function in the d-dimensional space. Search quality is measured by the fraction of the nearest K objects we are able to retrieve. Since feature-rich data objects are typically represented as high-dimensional feature vectors, similarity search is usually implemented as K-Nearest Neighbors (KNN) or Approximate Nearest Neighbors (ANN) search in high-dimensional feature-vector space where particular data structures could be maintained. It has been shown in [3] that the exact nearest neighbor problem suffers from the "curse of dimensionality", i.e. when the dimensionality exceeds about 10, existing data structures based on space partitioning are even slower than the brute-force, linear-scan approach.

In the real life, it is not necessary to insist on the perfect answer of returning K closest objects. Furthermore, determining an approximate answer should suffice [4, 5] and can reduce the search time tremendously. Even in some cases, an approximate algorithm can return the same result as a perfect algorithm. The objective of approximate nearest-neighbor indexing techniques is to find points whose distance from the query point is at most $(1+\varepsilon)$ times the exact distance of K nearest neighbors.

1.1 LSH Technique

Locality sensitive hashing (LSH) [6] technique is one of the best-known indexing methods for high-dimensional similarity search. LSH hashes data objects using multiple hash functions to ensure that, for each function, the probability of collision is much higher for objects which are close to each other than for those which are far apart. Although LSH-based methods have been shown to be successful [6–9], the basic LSH suffers from the following limitations.

- To achieve high search accuracy, the LSH method needs to use multiple hash tables to produce a good candidate set. Experimental studies show that the basic LSH method needs over a hundred [7] and sometimes several hundred hash tables [8] to achieve good search accuracy for high-dimensional data. Since the size of each hash table is proportional to the number of data objects, the basic LSH does not satisfy the space efficiency requirement.

- Since basic LSH is designed to guarantee the worst-case behavior, it might not be efficient on real-world data, which normally exhibits a rather "benign" behavior [2]. For example, some data points having locality characteristic typically form clusters, rather than being uniformly distributed in the space. Unfortunately, the basic LSH algorithm partitions the space uniformly and thus it does not exploit the clustering property of the data, which may result in slow query response and wasted space storing more hash tables.

The above limitations have been partially solved in previous work [10–12]. Not much work has been done to address the clustering and locality property of data, which can

be commonly observed in P2P file systems, in the similarity search using the LSH-based methods.

In this paper, we focus on the high-dimensional similarity search in the context of P2P file systems, exploiting data clustering characteristics. We make the main contributions as follows:

- We propose Bounded LSH (B-LSH), a simple and efficient method, to improve the accuracy of quick query response and space efficiency of high-dimensional similarity search in P2P file systems, where data often display the characteristics of locality and clustering that are rarely touched by previous work. The basic idea of *bounded* LSH is to set a maximum bucket size, derived from theoretical analysis, to guarantee approximate number of data objects in each bucket.

- We present practical algorithms associated with B-LSH, including *Initialization, Free Hashing, Local Ranking, Ordered Overflowing* and *Adjacent-Probe Searching*, to map non-uniform data objects into load-balanced buckets and further carry out efficient similarity search. By probing multiple buckets in a less number of hash tables, Bounded-LSH method will check far fewer buckets than previous basic-LSH method, which improves the query efficiency.

- We evaluated our method by using synthetic and real-world datasets on bounded LSH prototype in terms of search quality, accuracy and space efficiency. Experimental results show feasibility and efficiency of our proposed method. Compared to basic LSH, Bounded-LSH can successfully decrease the number of ranked objects by a factor of up to 4.26, and reduce the number of hash tables by a factor of up to 15.

The rest of the paper is organized as follows. Section 2 introduces the basic LSH algorithm, its key idea and structure. Section 3 presents our proposed bounded LSH method and practical operations. We evaluate the bounded LSH method using synthetic and real-world datasets in Section 4. Section 5 shows related work and we conclude our paper in Section 6.

2 LSH for Similarity Search

Locality Sensitive Hashing (LSH) is able to map similar objects, represented by feature vectors, into the same hash buckets with high probability. One similarity search query needs first to hash the query point q into buckets of multiple hash tables, located by LSH functions, and then union all objects existing in those chosen buckets and further rank the candidate objects according to their distances to query point q. In this section, we briefly describe the hash function and indexing method of basic LSH.

2.1 Hash Functions

Indyk and Motwani in [6] first introduced the idea of locality sensitive hashing to devise main memory algorithms

for nearest neighbor search. LSH function families have the property that objects that are close to each other have higher probability of colliding than objects that are far apart. Specially, let S be the domain of objects and D be the distance measure between two objects.

DEFINITION 1 *LSH function family, i.e.,* $\mathbb{H} = \{h : S \rightarrow U\}$ *is called* (r, cr, p_1, p_2)-*sensitive for distance function D if for any* $p, q \in S$

- *If* $D(p,q) \leq r$ *then* $Pr_{\mathbb{H}}[h(p) = h(q)] \geq p_1$,

- *If* $D(p,q) > cr$ *then* $Pr_{\mathbb{H}}[h(p) = h(q)] \leq p_2$.

To allow the similarity search, we choose $c > 1$ and $p_1 > p_2$. Thus, the problem of approximate nearest neighbor search can be solved by performing a series of hashing, searching and ranking within buckets of hash tables.

Different distance functions D, such as the well-known Jaccard measure, Hamming distance, ℓ_1 and ℓ_2 [6], can create different LSH families. LSH families for l_p norms, based on p-stable distribution [13], are proposed in [14] to allow each hash function $h_{a,b} : R^d \rightarrow Z$ to map a d-dimensional vector v onto a set of integers. The hash function is defined as: $h_{a,b}(v) = \lfloor \frac{a \cdot v + b}{W} \rfloor$ where a is a d-dimensional random vector with entries chosen independently from a p-stable distribution and b is a real number chosen uniformly from the range $[0,W]$.

2.2 Structure Construction

Constructing an LSH-based structure needs to determine two parameters: M, the capacity of a function family $\mathbb{G}$, and L, the number of hash tables.

- First, define a function family $\mathbb{G} = \{g : S \rightarrow U^M\}$ such that, for a d-dimensional vector v, $g(v) = (h_1(v), \cdots, h_M(v))$, where $h_j \in \mathbb{H}$ for $1 \leq j \leq M$. Thus, g is actually the concatenation of M LSH functions.

- Second, choose L functions $g_1, \cdots, g_L$ from $\mathbb{G}$ independently and uniformly at random. We further utilize each of L functions, g_i $(1 \leq i \leq L)$, to generate one associated hash table. Totally, we have L hash tables and a vector v will be hashed into a bucket (indicated by $g_i(v)$) in each hash table. Since the total number of hash buckets may be large, we only maintain non-empty buckets by using regular hashing [15, 16].

The optimal M and L values actually depend on the nearest neighbors' distance r. In practice, we use multiple sets of hash tables to cover different r values.

2.3 Basic Operations

Basic operations associated with LSH-based structures [6, 11] mainly include *Initialization*, *Insertion*, *Similarity Search* and *Deletion* for a vector v and our B-LSH also supports associated operations.

- **Initialization**: One LSH-based structure includes L hash tables, each of which contains M LSH functions in the form of $h_{a,b}(v) = \lfloor \frac{a \cdot v + b}{W} \rfloor$ using randomly chosen a and b.

- **Insertion**: A vector v needs to be inserted into L hash tables. To insert v into the i-th hash table, we first compute the hash value $g_i(v)$ and then place vector v into the bucket to which $g_i(v)$ points.

- **Similarity Search**: For a query point q, we compute the hash value $g_i(q)$ and find the bucket to which $g_i(q)$ points in the ith table. The chosen bucket contains part of candidate objects. We union all corresponding buckets in L hash tables as a candidate set, in which the objects need to be further ranked according to their distances to the query point q. Finally, we select the top K objects from the ranked set.

- **Deletion**: The operation for deletion is similar to insertion. We can remove vector v from the bucket to which $g_i(v)$ points.

2.4   Non-uniform Data Distribution

Newly emerging applications, e.g. P2P file systems, often exhibit their non-uniform data distribution with clustering property [17, 18]. Some of LSH buckets may contain too many data objects, which are close to each other according to the distance measure and thus, the overloaded buckets lead to the decrements of search accuracy and efficiency. One simple and naive solution is to carefully select an optimal r^* value that is used to evaluate the relative distance among data objects to guarantee that LSH buckets contain approximate number of data objects to obtain load balance in a hash table. However, performing the operation on selecting the optimal r^* value is cumbersome within a given dataset, even if using sampling method [10]. The sampling process requires knowledge of the nearest neighbor distance, which is difficult to observe in a data-dependent way.

3   Bounded LSH

Locality sensitive hashing and its variants are able to support similarity search in many practical applications where data are often assumed to be uniformly distributed. In this section, we focus on the design of simple and efficient algorithm and associated structure, called Bounded LSH (B-LSH), to facilitate similarity search for data, which are non-uniform distribution, such as Zipf or Gauss distribution. In fact, P2P file systems often need to handle similar "hot spot" data and Bounded-LSH can be used to improve system performance by exploiting access pattern of data locality. Figure 1 shows two examples of non-uniform distribution to illustrate the context of similarity search for the clustered data.
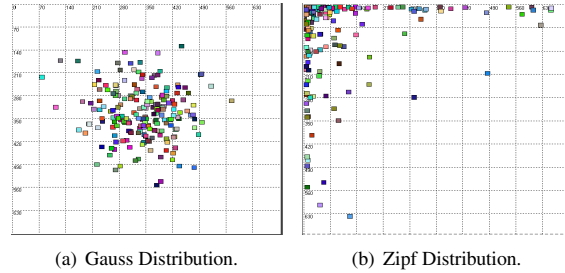

(a) Gauss Distribution.   (b) Zipf Distribution.

Figure 1. Two examples of typical non-uniform distribution.

3.1   Algorithm Overview

The key idea of our Bounded LSH scheme is to map the objects with non-uniform distribution into hash buckets to guarantee that the buckets contain approximate number of objects by bounding the size of each bucket in a hash table. Locality sensitive hashing [7] and its variants [10, 11] show that if the query object q near to an object p is not hashed to the same bucket as p, two buckets containing p and q are close to each other with a high probability. Our proposed Bounded LSH exploits this property to overflow extra objects into neighboring buckets, which can obtain space efficiency and query accuracy without sophisticatedly deriving the probing sequence [11] or carrying out sampling operations [10].

Basic LSH generally can not efficiently carry out similarity search for data objects with non-uniform distribution. Figure 2 shows two case studies where the query point q is far away from or close to the clustered data to satisfy top-3 similarity search, i.e., $K = 3$. We can observe that in Figure 2(a), it is very difficult to distinguish close data objects from clustered objects to select accurate candidates. Thus, we have to check many extra objects to guarantee search accuracy, decreasing query efficiency. A similar situation takes place where the query point q is close to the clustered objects, even if we only need to select 3 approximate nearest neighbors as shown in Figure 2(b). Unfortunately, we have to probe many extra objects, introducing substantial search costs.
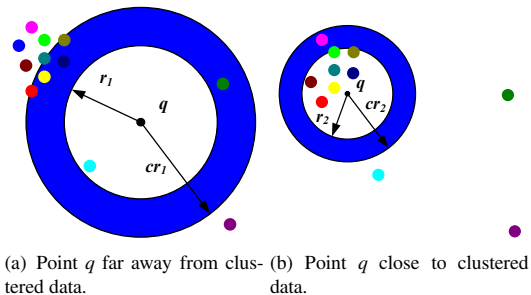

(a) Point q far away from clustered data.

(b) Point q close to clustered data.

Figure 2. Possible situations for a query point q in the 2-dimensional space.

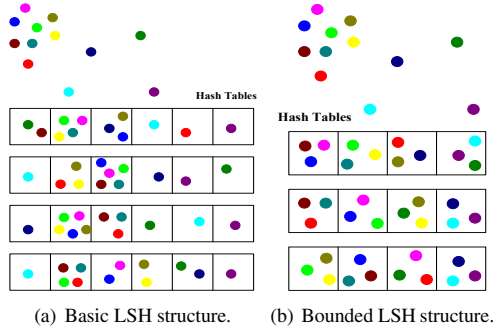(a) Basic LSH structure.　　(b) Bounded LSH structure.

Figure 3. Data storage using basic and bounded LSH structures.

Figure 3 shows the comparison between basic and bounded LSH structures containing the same number of clustered data objects. Basic LSH structure as shown in Figure 3(a) utilizes 4 hash tables to contain data objects based on hash computation regardless of their distribution. Thus, too many "hot spot" objects close to each other are possibly placed into the same or adjacent buckets that become overloaded, leading to very low search efficiency and accuracy due to the skewed data distribution. It is also observed that bounded LSH uses fewer hash tables than basic LSH since the buckets of bounded LSH are load-balanced and can obtain higher search accuracy.

Bounded LSH can map original non-uniformly clustered data objects to hash buckets, each of which contains approximate number of objects. The size of each bucket is important to design our bounded LSH. Space-efficient hashing algorithm in [12] allocates $(dn + n\log^{O(1)} n)$ space, which almost matches the lower bound for hash-based algorithm recently obtained in [19]. In the Bounded LSH, we define each bucket size as $\lceil (dn + n\log^{O(1)} n)/(L \cdot T) \rceil$ for n d-dimension objects stored in L hash tables, each of which maintains at most T buckets. In practice, choosing the parameters L and T that are optimal for a dataset needs first to determine the bounds on L and T that guarantee the design correctness and then, within those bounds, to choose the values L and T that can achieve the best expected trade-off between query running time and search accuracy[a].

3.2　Practical Operations

In this subsection, we present practical operations to be applied in B-LSH, including Initialization, Free Hashing, Local Ranking, Ordered Overflowing and Adjacent-Probe Searching. The first four operations assist the construction of B-LSH, while the last operation returns the similarity search results from it. Our proposed Bounded LSH can efficiently support similarity search for multi-dimensional data objects in the non-uniform distribution, meanwhile obtaining space efficiency and search accuracy.

---

[a]The implementation of Bounded LSH is based on basic LSH and uses simple methods for optimizing parameters [20].

Free Hashing for vector v

Compute $g_{i(v)}$ based on LSH functions
Insert v into the $Bucket_{g_i(v)}$

Figure 4. Free hashing algorithm to facilitate the insertion of vector v.

3.2.1　Initialization

Bounded LSH needs to construct L hash tables, each of which contains M LSH functions that are 2-stable Gaussian distribution for the Euclidean distance. Meanwhile, each bucket in a hash table can contain at most $\triangle_{B-LSH} = \lceil (dn + n\log^{O(1)} n)/(L \cdot T) \rceil$ objects.

Further operations on Bounded-LSH structure depend on the number of vectors existing in the bucket, i.e., $Number(Bucket_{g_i(v)})$, to which $g_i(v)$ points. A new arrival vector v needs to be inserted into L tables in the Bounded-LSH structure. Given the ith table, we will execute the associated operations as follows:

- $Number(Bucket_{g_i(v)}) < \triangle_{B-LSH}$: using *Free Hashing* to simply place vector v into the $g_i(v)$-th bucket in the i-th hash table.

- $Number(Bucket_{g_i(v)}) = \triangle_{B-LSH}$: Invoking *Local Ranking* to find a virtual center, $VC_{g_i(v)}$, of clustered vectors in that bucket and the maximum distance, called locality radius, $R_{g_i(v)}$, which is the longest distance coming from the center and edge point. Thus, we can obtain the *virtual center*, *locality radius* and *edge point*.

- $Number(Bucket_{g_i(v)}) > \triangle_{B-LSH}$: Requiring *Ordered Overflowing* to compute the distance of new vector v to the center. If the distance is larger than current locality radius, the new vector will be rejected. Otherwise, the new vector will be accepted and the original edge point is set as the excluded vector. The excluded vector will be sent to current bucket's neighbor and be treated as a new arrival.

3.2.2　Free Hashing

We can carry out *Free Hashing* on the bounded LSH when the bucket, which $g_i(v)$ points to, contains less than $\triangle_{B-LSH}$ objects. Consequently, B-LSH can easily place v into the $g_i(v)$-th bucket in the i-th hash table as shown in Figure 4.

3.2.3　Local Ranking

When the number of objects existing in a bucket is equal to the threshold $\triangle_{B-LSH}$, we need to execute the *Local Ranking* to find the virtual center, VC, whose d-dimensional coordinates are the average values of existing vectors (e.g., u) and the new arrival (i.e., v). We can further determine the locality radius between the center VC and one vector u_R, i.e., edge point, which has the longest distance away from the center in the bucket as shown in Figure 5.

Local Ranking for d-dimensional vectors

Compute $g_{i(v)}$ based on LSH functions
Insert v into the $Bucket_{g_i(v)}$
for $(x = 1, x \leq d, x++)$ **do**
$\quad VC^x_{g_i(v)} = \frac{\sum(u^x + v^x)}{Number(Bucket_{g_i(v)})}$
end for
$R_{g_i(v)} = distance(VC_{g_i(v)}, u_R)$

Figure 5. Local ranking algorithm to find the locality center and radius.

Ordered Overflowing in the bucket

Compute $g_{i(v)}$ based on LSH functions
if $distance(VC_{g_i(v)}, v) < R_{g_i(v)}$ **then**
$\quad$ Accept v and send the edge point (u_R) to neighboring
$\quad$ buckets of $Bucket_{g_i(v)}$
$\quad$ Execute *local ranking* in the bucket $Bucket_{g_i(v)}$
else
$\quad$ send v to neighboring bucket
end if

Figure 6. Ordered overflowing algorithm to obtain load balance among buckets.

3.2.4 Ordered Overflowing

Ordered overflowing can allow buckets in a hash table to store approximate number of vectors. When the number of vectors in a bucket is larger than $\triangle_{B-LSH}$, we need to first compute the distance between new vector point v and locality center VC, and then determine whether the new vector point should be excluded according to its distance to the virtual center, as shown in Figure 6. Specifically, if the distance is larger than $R_{g_i(v)}$, the bucket $Bucket_{g_i(v)}$ rejects v that is viewed as a new vector to the bucket's neighbors[b]. Otherwise, the current bucket can accept v and exclude the existing u_R that will be considered as a new vector to find one adjacent bucket accepting it. After accepting a new vector, the current bucket needs to carry out the *local ranking* to determine new locality center, radius and edge point.

3.2.5 Adjacent-Probe Searching

Multi-probe LSH proposed in [11] obtains significant space efficiency by using derived probing sequences to probe multiple hash buckets that are likely to contain query results in a hash table. However, multi-probe LSH needs to first analyze the bucket distance distribution of K nearest neighbors and then carefully generate a sequence of perturbation vectors, requiring more computations. Our method, adjacent-probe searching, is inspired by and further improves upon Multi-probe LSH by obtaining load balance of buckets, even for non-uniform distribution data objects, to further enhance space efficiency and search accuracy.

---

[b]The buckets may randomly choose left (right) adjacent neighbor. If the neighbor also rejects the vector, further right (left) non-repeated neighbor can continue to verify the vector until one bucket accepts v.
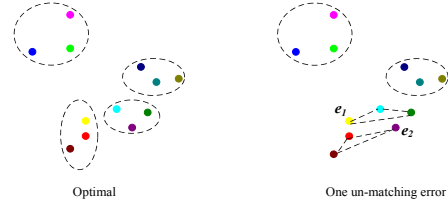

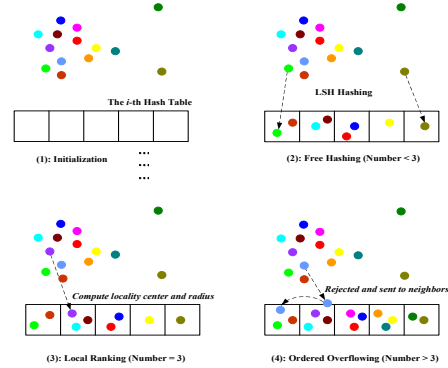**Figure 7. An example shows one possible un-matching error.**

Figure 8. Case study illustrates the construction operations of bounded LSH and $\triangle = 3$.

Adjacent-probe searching aims to probe more than one bucket for K approximate nearest neighbors when the bucket size is smaller than K. The LSH functions utilized are in the form of $h_{a,b}(v) = \lfloor \frac{a \cdot v + b}{W} \rfloor$. When the W value is reasonably large, it is observed [7, 11] that similar objects can be hashed into the same or adjacent buckets. Specifically, for a query point q requiring its K approximate nearest neighbors, we need to first probe the $g_i(q)$-th bucket and then check its $\lceil \frac{K}{\lceil (dn + n \log^{O(1)} n)/(L \cdot T) \rceil} \rceil - 1$ adjacent neighboring buckets in the i-th hash table that contains n d-dimensional objects and each bucket size is $\triangle_{B-LSH}$. Thus, we can further union the objects from the probed buckets and rank their distances to the query point q and finally return top-K objects, obtaining space efficiency and fast searching.

3.3 Case Study

We present a simple example to illustrate how our bounded LSH works and obtains space efficiency and search accuracy. Figure 8 shows the case study (for one hash table) that clearly describes how to construct the bounded LSH structure, which maintains load balance among multiple buckets by utilizing the proposed operations.

Figure 9 and 10 respectively show the similarity search requiring top-3 and top-4 approximate nearest neighbors. We can answer top-3 search by probing one bucket which $g_i(q)$ points to in the i-th hash table. Given different requests from query points q_1 and q_2, bounded LSH respec-
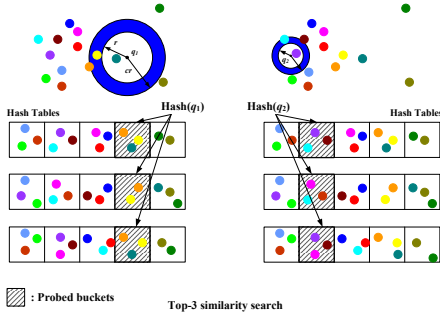
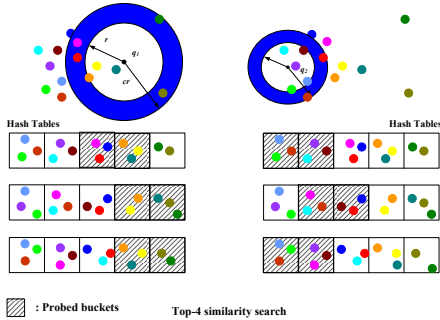**Figure 9. B-LSH for top-3 similarity search.**

Figure 10. B-LSH for top-4 similarity search.

tively checks 3 and 4 vectors after the candidate union operation from 3 buckets.

However, for top-4 search, one bucket obviously cannot satisfy the query requirement. Therefore, we need to randomly choose the $(g_i(q) - 1)$-th or $(g_i(q) + 1)$-th bucket and probe its vectors. As shown in Figure 10, bounded LSH needs to verify 8 vectors for the query points. In contrast to the basic LSH probe method, bounded LSH can significantly reduce the number of probed vectors to improve query efficiency while maintaining query accuracy. In the following section, we will further show the significant advantages of bounded LSH that has been implemented over basic LSH and further compare their performance in terms of space efficiency, search accuracy and query latency.

4 Performance Evaluation

In this section, we report the evaluation results by implementing the bounded LSH prototype in the Linux environment. We examine the performance of basic and bounded LSH methods in multiple metrics. Experiments were conducted using synthetic and real world datasets.

4.1 Experiment Setup

Our simulations run on 3.2GHz Dual Core processors with 2GB RAM. We generated the well-connected random graph as the P2P network topology by using GT-ITM topology generator[c].

---

[c]http://www.cc.gatech.edu/projects/gtitm/

Before we look at real-world examples, we first test our bounded LSH on artificial datasets, which follow Gaussian distribution generated by GSTD (Generating Spatio-Temporal Datasets) generator [21]. The data for experiments are horizontally partitioned evenly among peers. The results will help us to get a clear image as how the basic and bounded LSH algorithms perform. We further evaluated the bounded LSH for P2P similarity search by using a real-world dataset, Covtype[d], which is widely used in information indexing researches. The Covtype consists of 54-dimensional instances of 581012 forest Covertype data, available from UCI Machine Learning Repository.

For each tested dataset, we randomly pick 100 objects as the query objects. For each query, a peer initiator is randomly selected. The ideal result is K nearest neighbors of the query object based on the Euclidean distance of their feature vectors. To perform meaningful comparisons, we ran algorithms on random subsets of original data sets with size increased from 5% to 100%.

4.2 Implementation Details

We have implemented Bounded LSH based on the basic LSH method. For bounded LSH, we implemented all associated operations, including *Initialization, Free Hashing, Local Ranking, Ordered Overflowing* and *Adjacent-Probe Searching*. To save more space, we only store object IDs in the hash buckets and associated vectors accessed via object IDs are stored in a separate data structure. We also maintained a simple object ID bitmap to explicitly indicate the chosen objects in different hash buckets and the bitmap can help to efficiently union objects to facilitate further ranking operations. To evaluate the query accuracy, we need to know the optimal query results by executing the brute-force method, which linearly scans through all feature vectors to find the K nearest objects.

Since the distance variance in a dataset in most cases does not depend on the specific query point but on the intrinsic properties of the dataset [22], we can choose the same parameter r to test a vast majority of queries. To understand the scalability of our algorithms, we did run experiments respectively to derive approximate 10 and 20 nearest neighbors. We varied different parameter values for the basic and bounded LSH methods. The default values are $W = 0.6$, $M = 10$ for the synthetic dataset and $W = 18$, $M = 12$ for the real-world dataset. Initially, LSH method needs to read all feature vectors into the main memory. Since an entry in each hash table consumes about 16 bytes in our implementation, 2 GB main memory can contain the index data structure of the LSH-based methods.

4.3 Result Analysis

We select a set of cost metrics to comprehensively evaluate the performance. They include *Number of Ranked Objects*, *Recall*, *Search Latency* and *Number of Hash Tables*.

---

[d]http://www.ics.uci.edu/ mlearn/MLRepository.html

4.3.1 Number of Ranked Objects

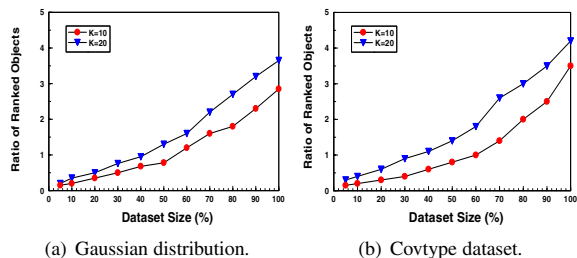(a) Gaussian distribution.　　(b) Covtype dataset.

Figure 11. Ratio of ranked objects.

Figure 11 shows the experimental results in terms of the number of ranked objects for similarity search with $K = 10$ and $K = 20$. We focus on the ratio of ranked objects of the basic LSH to bounded LSH. We randomly used the subsets of the whole dataset as the input to evaluate the number of ranked objects that come from the union operation on hash buckets, which are indicated by the hash values of query points. With the increment of data, we observe that bounded LSH can significantly decrease the number of ranked objects by factors of up to 3.65 and 4.26 respectively for Gaussian distribution and real Covtype dataset when carrying out $K = 20$ similarity search as shown in Figure 11(a) and 11(b).

4.3.2 Recall

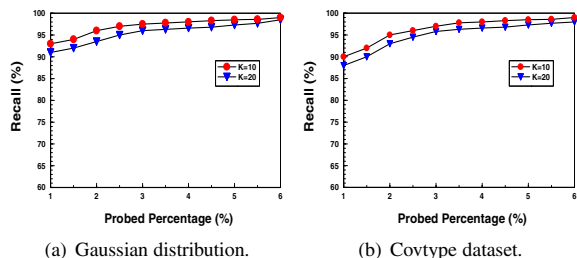(a) Gaussian distribution.　　(b) Covtype dataset.

Figure 12. Recall.

We examine the *Recall* of bounded LSH to evaluate its search quality when increasing the percentage of probed objects to the whole dataset. Figure 12 shows that the search quality for $K = 10$ and $K = 20$ nearest neighbors is not sensitive to the parameter K under different percentages of probed objects to those of the whole dataset. In particular, the K-aware sensitivity decreases as the increment of the number of examined objects since those two curves gradually merge.

4.3.3 Search Latency

Figure 13 shows the search latency when executing $K = 20$ similarity search. It can be observed that bounded LSH obtains significant advantage over basic LSH, saving up to 42.6% search time. The main reason is that bounded LSH
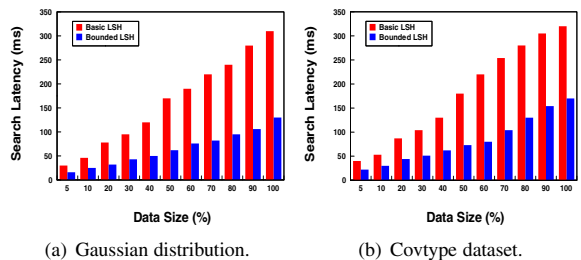

(a) Gaussian distribution.　　(b) Covtype dataset.

Figure 13. Search latency.

decreases the number of ranked objects from selected hash buckets and thus improves the search efficiency.

4.3.4 Number of Hash Tables

The results in Figure 14 show that the bounded LSH method for carrying out the indexing of $K = 20$ nearest neighbors is significantly more space efficient than the basic LSH method. For both the synthetic and real-world datasets, the bounded LSH method reduces the number of hash tables by a factor of up to 15 given similar query time.
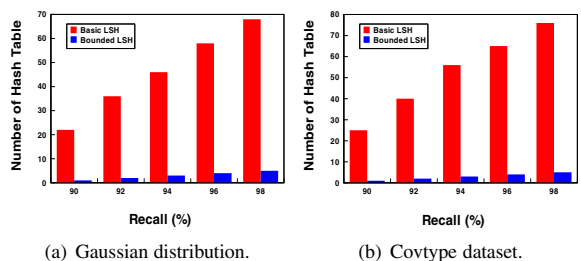

(a) Gaussian distribution.　　(b) Covtype dataset.

Figure 14. Number of required hash tables.

5 Related Work

Similarity search is usually implemented as K nearest neighbors or approximate nearest neighbors search in high-dimensional feature-vector space. Recent work includes distributed nearest neighbor-based condensation of very large data sets [23], dominating queries [24], stream time series [25] and simple and discriminative typicality queries [26]. Due to space limitation, we ignore some algorithms for approximate nearest neighbor search and readers can refer to the survey in [2].

Locality sensitive hashing (LSH) [6] and its variations have been proposed as indexing techniques for approximate similarity search in metric spaces. Although LSH algorithm enjoys a rigorous, theoretical performance guarantee, the basic LSH has its own limitations and some recent work makes improvements on them. Entropy-based LSH method [10] is proposed to generate randomly "perturbed" objects near the query point, further search objects, and finally return the union of all results as the candidate set. Multi-Probe LSH [11] intelligently probes multiple buckets that are likely to contain query results in a hash table. It

has a similar time efficiency as the basic LSH method while reducing the number of hash tables by an order of magnitude. Near-optimal hashing algorithms in [12] can guarantee a space-efficient version using $(dn + n\log^{O(1)} n)$ space and a query time of $(dn^{O(1/c^2)})$.

6 Conclusion

This paper presents the bounded LSH method for similarity search, which takes the data locality into consideration in P2P file systems. The bounded LSH maps objects displaying non-uniform distribution property into hash buckets with bounded capacity to obtain load balance and space efficiency. Our experimental results using real and synthetic datasets show that the bounded LSH, compared to the basic LSH, can decrease the number of ranked objects by a factor of up to 4.26 and reduce the number of hash tables by a factor of up to 15. Although bounded LSH shows significant advantages over basic LSH, a comparison of bounded LSH and other variants of LSH-based methods would also be helpful especially when we evaluate the implementations in large-scale P2P file systems. We plan to study these issues in the future work.

References

[1] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. E. Abbadi, "High Dimensional Nearest Neighbor Searching," *Elsevier Information Systems Journal*, September 2006.

[2] T. Liu, A. Moore, A. Gray, and K. Yang, "An investigation of practical approximate nearest neighbor algorithms," *Proceedings of Neural Information Processing Systems*, vol. 1, pp. 1–5, 2004.

[3] R. Weber, H. Schek, and S. Blott, "A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces," *VLDB*, pp. 194–205, 1998.

[4] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. Ullman, and C. Yang, "Finding interesting associations without support pruning," *TKDE*, vol. 13, no. 1, pp. 64–78, 2001.

[5] P. Indyk, G. Iyengar, and N. Shivakumar, "Finding pirated video sequences on the Internet," *Technical Report, Computer Science Department, Stanford University*.

[6] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," *STOC*, pp. 604–613, 1998.

[7] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," *VLDB*, pp. 518–529, 1999.

[8] J. Buhler, "Efficient large-scale sequence comparison by locality-sensitive hashing," *Bioinformatics*, vol. 17, no. 5, pp. 419–428, 2001.

[9] X. Yang and Y. Hu, "A Landmark-based Index Architecture for General Similarity Search in Peer-to-Peer Networks," *IPDPS*, 2007.

[10] R. Panigrahy, "Entropy based nearest neighbor search in high dimensions," *SODA*, pp. 1186–1195, 2006.

[11] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search," *VLDB*, 2007.

[12] A. Andoni and P. Indyk, "Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions," *FOCS*, pp. 459–468, 2006.

[13] P. Indyk., "Stable distributions, pseudorandom generators, embeddings, and data stream computation," *FOCS*, pp. 189–197, 2000.

[14] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," *ACM Symposium on Computational Geometry*, pp. 253–262, 2004.

[15] R.Motwani and P.Raghavan, "Randomized Algorithms," *Cambridge University Press*, 1995.

[16] M.L.Fredman, J.Komlos, and E.Szemeredi, "Storing a sparse table with O(1) worst case access time," *Journal of the ACM*, pp. 538–544, 1984.

[17] B. Yang and H. Garcia-Molina, "Improving search in peer-to-peer networks," *ICDCS*, pp. 5–14, 2002.

[18] K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *IEEE Communications Surveys & Tutorials*, pp. 72–93, 2005.

[19] R. Motwani, A. Naor, and R. Panigrahy, "SLower bounds on locality sensitive hashing," *ACM Symposium on Computational Geometry*, 2006.

[20] A. Andoni and P. Indyk, "E2LSH: Exact euclidean locality-sensitive hashing," *Implementation available at http://web.mit.edu/andoni/www/LSH/index.html*.

[21] Y. Theodoridis, J. Silva, and M. Nascimento, "On the Generation of Spatiotemporal Datasets," *Proc. SSD*, pp. 147–164, 1999.

[22] P.Ciaccia, M.Patella, and P.Zezula, "A cost model for similarity queries in metric spaces," *PODS*, pp. 59–68, 1998.

[23] F. Angiulli and G. Folino, "Distributed Nearest Neighbor-Based Condensation of Very Large Data Sets," *TKDE*, vol. 19, no. 12, pp. 1593–1606, 2007.

[24] M. L. Yiu and N. Mamoulis, "Efficient Processing of Top-k Dominating Queries on Multi-Dimensional Data," *VLDB*, 2007.

[25] X. Lian, L. Chen, and B. Wang, "Approximate Similarity Search over Multiple Stream Time Series," *DASFAA*, 2007.

[26] M. Hua, J. Pei, A. W. C. Fu, X. Lin, and H.-F. Leung, "Efficiently Answering Top-k Typicality Queries on Large Databases," *VLDB*, 2007.