Locality-Sensitive Bloom Filter for Approximate Membership Query

Yu Hua, *Member*, *IEEE*, Bin Xiao, *Senior Member*, *IEEE*,
Bharadwaj Veeravalli, *Senior Member*, *IEEE*, and Dan Feng, *Member*, *IEEE*

Abstract—In many network applications, Bloom filters are used to support exact-matching membership query for their randomized space-efficient data structure with a small probability of false answers. In this paper, we extend the standard Bloom filter to Locality-Sensitive Bloom Filter (LSBF) to provide Approximate Membership Query (AMQ) service. We achieve this by replacing uniform and independent hash functions with locality-sensitive hash functions. Such replacement makes the storage in LSBF to be locality sensitive. Meanwhile, LSBF is space efficient and query responsive by employing the Bloom filter design. In the design of the LSBF structure, we propose a bit vector to reduce False Positives (FP). The bit vector can verify multiple attributes belonging to one member. We also use an active overflowed scheme to significantly decrease False Negatives (FN). Rigorous theoretical analysis (e.g., on FP, FN, and space overhead) shows that the design of LSBF is space compact and can provide accurate response to approximate membership queries. We have implemented LSBF in a real distributed system to perform extensive experiments using real-world traces. Experimental results show that LSBF, compared with a baseline approach and other state-of-the-art work in the literature (SmartStore and LSB-tree), takes less time to respond AMQ and consumes much less storage space.

Index Terms—Approximate membership query, bloom filters, locality sensitive hashing.

1 Introduction

In many real-world and large-scale network applications, it is more attractive and interesting to support *Approximate Membership Query* (AMQ), i.e., "$q \rightarrow S$?," rather than *exact-matching membership query*, i.e., "$q \in S$?." By transforming "$q \in S$?" to "$q \rightarrow S$?," we do not need to probe the presence of point q but its proximity to any member in the set S under a given metric. Existing data ocean makes exact-matching queries costly that are too fragile and sensitive to data inconsistency and staleness. The brute-force searching to respond the exact-matching query can cause extremely high cost. In contrast, AMQ can relax constraints on user request to take much shorter time for the user to get satisfactory results. It also helps to identify uncertain and inaccurate inputs. In fact, the exact and approximate membership queries are not a new problem [1] but become crucial to existing and possible future applications due to their wide applications and performance requirements for query optimization that is unfortunately not fully addressed by conventional approaches. Recent research in [2], and [3] reveals that improvements made to AMQ can benefit both users and system performance.

- *Y. Hua and D. Feng are with the School of Computer Science and Technology, Wuhan National Lab for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {csyhua, dfeng}@hust.edu.cn.*
- *B. Xiao is with the Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong. E-mail: csbxiao@comp.polyu.edu.hk.*
- *B. Veeravalli is with the Department of Electrical and Computer Engineering, The National University of Singapore, 4 Engineering Drive 3, Singapore 117576. E-mail: elebv@nus.edu.sg.*

Manuscript received 3 May 2010; revised 23 Dec. 2010; accepted 11 May 2011; published online 14 June 2011.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2010-05-0269.
Digital Object Identifier no. 10.1109/TC.2011.108.

AMQ aims to determine whether a given query q is *approximate* to a data set S. Specifically, given a d-dimensional metric space U represented as (U, d), let S be the set of points in this space and $S \subset U$. Given a constant parameter R, the query point q is accepted as an approximate member if $\exists p \in S$, the pair distance is true for $\|p, q\| \leq R$. Furthermore, the points approximate to set S essentially constitute a set S' that is a superset of set S, i.e., $S \subseteq S'$. The approximate members of set S are the exact members of set S'.

1.1 Motivations

In query service tools, standard Bloom filters have been very powerful for the compact set representations with low false positive and negative probabilities [4]. The $O(1)$ complexity in Bloom filters to support fast exact-membership query involves simple hashing, setting, and testing "0/1" bits in a bit vector. Their space-efficient structures hence have been widely used in many network applications, such as heavy flow identification [5], content summary [6], optimal replacement [7], the longest prefix matching [8], route lookup [9], and packet classification [10]. Unfortunately, standard Bloom filters are unable to support AMQ and ignore the potential hits of proximate items, since they use uniform and independent hash functions (e.g., MD5 and SHA-1) to provide boolean-based answers.

The Locality Sensitive Hashing (LSH) [11] technique paves the way for solving AMQ. LSH can faithfully keep the locality of items in a data set by mapping similar items into the same hash bucket with a high probability. However, to get the AMQ result, LSH needs to use L locality hash functions to store a member L times into L hash tables, which is space consuming.

Yet, we do not have a single space-efficient data structure to accurately support AMQ. Although some existing work proposed the use of Bloom filters in the AMQ, such as distance-sensitive Bloom filters [12] and optimal Bloom filter replacement [7], they did not provide the detailed data structure design using LSH for real implementations. Nonetheless, they did not take into account the potential false answers from false positives and negatives. The false answers often occur in large-scale (e.g., EB 10^{18} bytes or ZB 10^{21} bytes data) distributed database systems due to data inconsistency and staleness. Our work is motivated by real application requirements and makes further improvements upon above work by extending data attributes to high dimensions and using simple but efficient verification schemes to reduce false answers.

1.2 Our Contributions

In this paper, we propose a novel structure, Locality-Sensitive Bloom Filter (LSBF), to efficiently support fast AMQ without compromising query performance. LSBF is a space-efficient structure using bit-wise vectors. It functions as normal LSH to hash an item to buckets where a bucket is a binary bit, from which a bit vector can indicate the existence of proximate items. The design is based on the observation that Bloom filters can map original items into a relatively succinct storage space with the aid of variant hash functions. Therefore, it is feasible to replace independent and uniform hash functions in Bloom filters with LSH functions while maintaining item proximity (due to LSH property) and storage-space efficiency in a bit vector (due to Bloom filter property). Moreover, when facing query errors caused by Bloom filters and LSH functions, we propose new verification schemes in LSBF to dramatically improve query accuracy. Through theoretical analysis and extensive experiments on distributed system implementations, we show the efficiency of LSBF to handle AMQ in terms of quick query response, high query accuracy, low I/O cost, and space overhead. Our contributions are summarized as follows:

- We propose an LSBF structure that replaces conventional random and independent hash functions with locality sensitive hash functions [11] to measure locality of items and support AMQ. The query time is $O(dL) + O(k)$ complexity and the space overhead is $O(dn/\omega) + O(n \cdot k/\ln 2)$ bits when we store n items with d-dimensional attributes in LSBF, using L locality-sensitive hash functions, a predefined interval size ω in a projected line, and k uniform hash functions in a standard Bloom filter.
- We present a bit-based verification scheme by adopting a verification Bloom filter to significantly reduce False Positives (FP) and an active overflowed scheme to decrease False Negatives (FN), respectively. These schemes are critical for improving query accuracy because approximate queries in LSBF can cause FP and FN due to hash collisions and the probabilistic hashing property in LSH. Specifically, a nonapproximate member may be viewed as a member in an FP when its associated bits are hashed multiple times by others. Nevertheless, LSH can hash proximate items into neighboring bits with some probability, which may result in an FN answer to a real approximate item.
- We give rigorous theoretical analysis of LSBF for its FP, FN probability, and space overhead. To verify LSBF in real applications, we implement the proposed LSBF and examine its performance through real-world traces representing operations on file systems and high-dimensional environmental data. The performance has been compared with other state-of-the-art work to show its fast query accuracy and space efficiency.

The rest of the paper is organized as follows: Section 2 shows the AMQ problem and some backgrounds. We present the design of LSBF in Section 3. Section 4 shows theoretical analysis of the proposed LSBF in terms of query accuracy, time, and space complexity. We carry out performance evaluation of LSBF in Section 5. Section 6 gives examples of using LSBF structure in real-world applications and Section 7 shows related work. We conclude our paper in Section 8.

2 AMQ Problem and Backgrounds

In this section, we first define the AMQ problem. We then show the backgrounds of Bloom filter and LSH functions that are key components in our design of LSBF.

2.1 Problem Description

We propose LSBF to maintain locality of items in a data set to support AMQ. Given an item q, LSBF needs to determine whether it is approximate to any item in a data set S in a constrained metric by examining the distance measured by l_s norm where the metric can be Hamming or euclidean distance [13]. We use $\|*\|$ to denote the measured distance between two items in a d-dimensional space. Now, we present the AMQ problem.

Problem 1 (Approximate Membership Query). *Given a parameter R, a queried item q is regarded as an approximate member of a data set S if $\exists p \in S$, $\|p, q\| \leq R$.*

Problem 2 (c-Approximate Membership Query). *Given parameters c and R and a query point q, data set S accepts q as a c-approximate member if $\exists p \in S$, $\|q, p\| \leq cR$ and $c \geq 1$.*

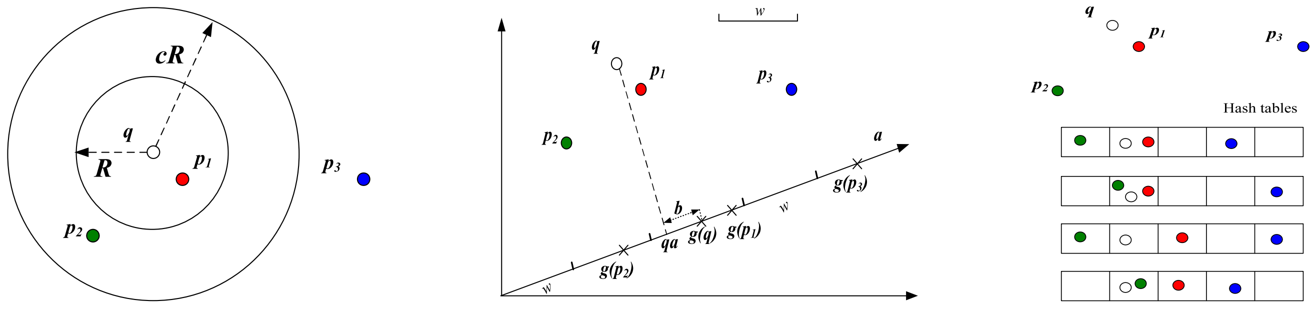
The AMQ problem is the special case of the *c-approximate Membership Query* problem when we set $c = 1$, which is the focus of this paper. In addition, LSBF may cause both false positives and negatives for AMQ due to the used Bloom filter and locality sensitive hashing functions.

Definition 1 (False Positive of AMQ). *A queried item q is a false positive to data set S if the query receives positive answer while in fact $\forall p \in S$, $\|p, q\| > R$ for a given parameter R.*

Definition 2 (False Negative of AMQ). *A queried item q is a false negative to data set S if the query receives negative answer while in fact $\exists p \in S$, $\|p, q\| \leq R$ for a given parameter R.*

The false positives and negatives of the c-approximate membership query can be defined similarly by replacing R with cR.

To alleviate false positives and negatives and ensure high query accuracy of AMQ, we propose simple computation to verify the existence of proximate items as shown in Section 3. Note that if q is in fact a member of S, AMQ becomes the exact membership query that can be supported by standard Bloom filters. Although our LSBF can also support conventional exact membership query, this paper



(a) Illustration of measured distance. (b) Geometry result of hash functions. (c) Storage form.

Fig. 1. An example of LSH scheme to hash proximate points into the same bucket in hash tables with a high probability.

only focuses on the response to AMQ using compact Bloom filters, which has rarely been addressed by previous work.

2.2 Key Components

This section mainly describes backgrounds of Bloom filters and LSH functions, which will be used in our LSBF design.

2.2.1 Bloom Filter

A standard Bloom filter is a bit array of m bits for representing a data set $S = \{a_1, a_2, \dots, a_n\}$ of n items. All bits in the array are initially set to 0. Then, a Bloom filter uses k independent hashing functions $\{h_1, \dots, h_k\}$ to map the set to the bit vector $[1, \dots, m]$. Each hash function h_i maps an item a to one of the m-array positions with uniform random distribution. To determine whether an item a is an exact member of set S, we need to check whether all $h_i(a)$ are set to 1. Otherwise, a is not in the set S. The membership query in a Bloom filter possibly introduces a false positive, indicating that an item a is a members of set S although it in fact is not. The false positive rate of standard Bloom filter is $f_{StandardBF} \approx (1 - e^{-\frac{kn}{m}})^k$ when the Bloom filter has m bits and k hash functions for storing n items. The probability can obtain the minimum $(1/2)^k$ or $(0.6185)^{m/n}$ when $k = (m/n)\ln 2$.

From the standard Bloom filter, many variants are investigated, including counting Bloom filters [14], compressed Bloom filters [15], group-hierarchical Bloom filter array [16], space-code Bloom filters [17], spectral Bloom filters [18], multidimension dynamic Bloom filters [19], parallel Bloom filters [20], load balanced Bloom filters [21], combinatorial Bloom filters [22], and incremental Bloom filters [23]. Other details can be referred to the survey of Bloom filters [24].

2.2.2 Locality Sensitive Hashing

Locality Sensitive Hashing introduced by Indyk and Motwani in [11] maps similar items into the same hash buckets with a high probability to serve main memory algorithms for similarity search. Then, for a given request for similarity search query, we need to hash query point q into buckets in multiple hash tables, and furthermore unite all items in those chosen buckets by ranking them according to their distances to the query point q. We can hence select closest items to a queried one. LSH function family has the property that items that are close to each other will have a higher probability of colliding than items that are far apart. We define S to be the domain of items and $\|*\|$ to be the distance metric between two items.

Definition 3. *LSH function family, i.e., $\mathbb{H} = \{h : S \rightarrow U\}$ is called (R, cR, P_1, P_2)-sensitive for distance function $\|*\|$ if for any $p, q \in S$*

- *If $\|p, q\| \leq R$ then $Pr_{\mathbb{H}}[h(p) = h(q)] \geq P_1$.*
- *If $\|p, q\| > cR$ then $Pr_{\mathbb{H}}[h(p) = h(q)] \leq P_2$.*

To allow the similarity search, we choose $c > 1$ and $P_1 > P_2$. In practice, we need to enlarge the gap between P_1 and P_2 by using multiple hash functions. Distance functions $\|*\|$ correspond to different LSH families of l_s norms based on s-stable distribution 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 in $\mathbb{H}$ can be defined as:

$$h_{a,b}(v) = \left\lfloor \frac{a \cdot v + b}{\omega} \right\rfloor, \tag{1}$$

where a is a d-dimensional random vector with chosen entries following an s-stable distribution and b is a real number chosen uniformly from the range $[0, \omega)$ where ω is a large constant.

Fig. 1 shows an example to illustrate the LSH working scheme in terms of measured distance, geometry result of hash functions, and the storage form of hash tables. Specifically, LSH can determine the proximate locality between two points by examining their distance in a metric space. If the circle centered at q with radius R covers at least one point, e.g., p_1, as shown in Fig. 1a, LSH can provide a point with no more than cR distance to q as a query result. We can observe that there is an uncertain space in LSH from R to cR distance and the query q will obtain a reply of either point p_1 or p_2, since both points locate within distance cR, i.e., $\|p_1, q\| < cR$ and $\|p_2, q\| < cR$. On the other hand, point p_3 is not close to the queried q due to its distance larger than cR.

Fig. 1b further exhibits the geometry result of locality-sensitive hash functions in a 2D space. Given a vector a and query point q, $q \cdot a$ is the dot product of them. We uniformly choose b from the interval $[0, \omega)$. We can observe that $q \cdot a$ is the projection of point q onto vector a, denoted as $h(q)$. From it, we get $g(q)$ with a shifted distance b. Since the vector a line is divided into intervals with length ω, each interval corresponds to the position sequence number of point q. In such transformation, proximate points, e.g., q and p_1, have a high probability to be located into the same interval.

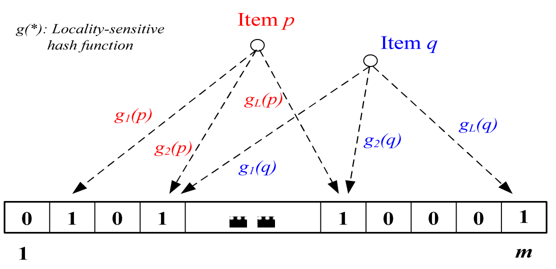


Fig. 2. Basic LSBF in the bit-vector implementation.

LSH implements the locality-sensitive approach by using multiple hash tables as shown in Fig. 1c, in order to produce a higher probability of containment within one bucket for proximate items. Since query point q is a close neighbor to point p_1, they are stored into one bucket of hash tables with a high probability. For instance, they are in the same bucket in the first and second hash tables in Fig. 1c. In contrast, point p_3 has a very low opportunity to locate together with point q into one bucket due to their long euclidean distance. In addition, LSH exhibits its approximate property by presenting the uncertain location for point p_2 because p_2 locates between R and cR.

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, we define a function family $\mathbb{G} = \{g : S \rightarrow U^M\}$ such that, for a d-dimensional vector v, $g(v) = (h_1(v), \ldots, h_M(v))$, where $h_j \in \mathbb{H}$ for $1 \leq j \leq M$. $g(v)$ hence becomes the concatenation of M LSH functions. Second, we randomly select L functions $g_1, \ldots, g_L$ from $\mathbb{G}$, each of which, $g_i(1 \leq i \leq L)$, is associated with one hash table, thus requiring L hash tables. A vector v will be further hashed into a bucket (positioned by $g_i(v)$) in each hash table. Since the total number of hash buckets may be large, we can only maintain nonempty buckets by using the regular hashing in a real implementation. The optimal M and L values actually depend upon the definition of nearest neighbors' distance R. In practice, we use multiple sets of hash tables to cover different R values.

3 Locality Sensitive Bloom Filter

In this section, we first show the basic LSBF structure and explain reasons for false positives and negatives of query results. To ensure high accuracy for AMQ, we, respectively, present the bit-based verification scheme to alleviate false positives and the active overflowed scheme to alleviate false negatives in the basic LSBF. Thus, the LSBF structure with extra schemes can accurately support AMQ while obtaining significant space savings and providing quick query response by using Bloom filters and locality sensitive hashing computation.

3.1 Basic LSBF Structure

A locality-sensitive Bloom filter consists of an m-bit array where each bit is initially set to 0. There are totally L locality-sensitive hash functions, $g_i(1 \leq i \leq L)$, to hash an item into bits, rather than its original buckets in hash tables, to significantly decrease the space overhead. An item as an input of each hash function g_i is mapped into a bit based on the hash computation. A bit is hence possibly set to one more than once and only the first setting takes effect. All items belonging to a data set S can be inserted into the m-bit array space that then serves as a summary vector of the data set S to support approximate queries. When an approximate query request for item q arrives, we execute the same operations to insert an item by hashing $g_i(q)(1 \leq i \leq L)$ to L bits. If all L bits are "1," we determine the item q is an approximate member of the data set S in the metric R, i.e., $\exists\, p \in S, \|p, q\| \leq R$. Fig. 2 shows an example to illustrate the proposed LSBF structure when we insert two items p and q.

In LSBF, a positive answer should be returned when a queried item is within the distance R to an existing one in a data set S. Fig. 3 shows the geometry description of query results from the LSH computation. Given 2D vectors, a and b, Fig. 3a shows the approximate query where the queried point q is covered (the radius to be R) by an existing item p_1 in the data set S. We then probe the corresponding bits in LSBF by checking the $g_a(q)$ and $g_b(q)$, both of which should be "1" that have been set by item p_1. The point q is hence viewed as an approximate member of data set S.

While inheriting the benefits of Bloom filters for fast query and space saving, LSBF has to deal with the potential false positives. Conventional false positives in Bloom filters for an exact-matching membership query say an item is a member of a data set while it in fact is not. Therefore, the false positive of exact-matching query in a standard Bloom filter is essentially a boolean decision. On the other hand, for an AMQ in LSBF, we need to calibrate the conventional false positive to be that a queried item q is falsely viewed as an approximate member of a data set S while in fact $\forall p \in S$, $\|p, q\|_s > R$, in the s-stable metric space. Therefore, the approximate membership only exhibits a relative relationship and heavily depends upon the parameter R and the used s-stable metric space. Note that when the context is clear, we ignore the subscript s.

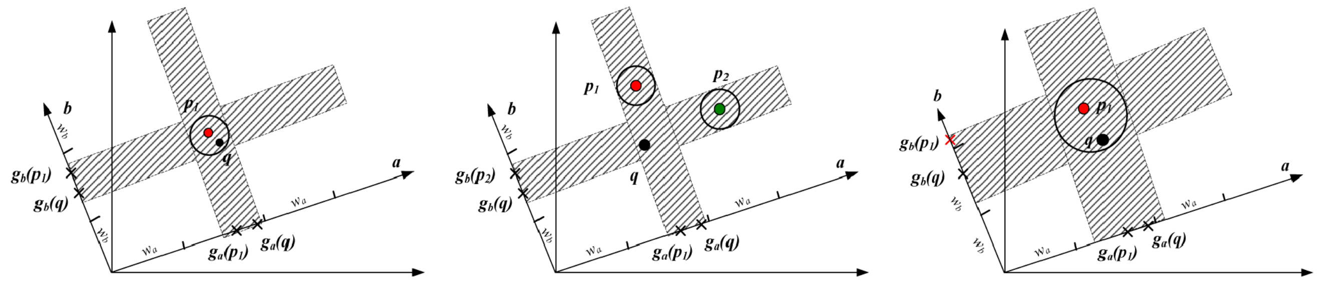


Fig. 3. Geometry description of correct and false answers due to the probabilistic hash collision property of LSH. (a) Queried point q covered by one existing item p_1 in a data set S. (b) False positives from multidimensional inconsistency. (c) False negatives from multidimensional checking.

3.2 False Positive and False Negative

Given a certain parameter R, LSBF possibly produces both false positives and false negatives for AMQ. False positives are the results of potential hash collisions and lack of identity consistency verification in terms of multidimensional attributes from distinct items. Multiple hashes in LSBF may give rise to hash collisions where a queried item can be wrongly treated as an approximate member but in fact its hashed bits are set by other items.

Besides the hash collision, multidimensional attributes of an item may also cause false positives. For example, the inconsistency of checked multidimensional attributes may produce false positives as shown in Fig. 3b. Although the queried point q is not covered neither by point p_1 nor by p_2, point q is still considered as an approximate member of the queried data set S because in the m-bit array, both hashed bits are "1." The main reason for false positive is the loss of the union of multidimensional attributes to one identity and we can only determine that the queried point q is approximate to the data set in *each* dimension. Unfortunately, LSBF is unable to tell whether the approximate membership in each dimension comes from an existing item or from multiple items. We need to develop a verification scheme to reduce such kind of false positives.

On the other hand, false negatives essentially come from the probabilistic property of locality-sensitive hashing functions that can hash proximate items into the same bit with a high, but *not 100 percent*, probability. Two close-by items thus may be hashed into the same, adjacent or even remote bits. Fig. 3c shows an example to illustrate the false negative in LSBF where two close points p_1 and q are mapped to the same bit (or the same interval) when projected onto vector a. However, they are mapped to adjacent bits (different intervals) for vector b. The approximate query for point q obtains "1" in bit $g_a(q)$ but "0" in bit $g_b(q)$, implying that the bit $g_b(q)$ is not previously set by other items in the data set S. Since it is not all bits "1," q is not regarded as an approximate member although it is, thus producing a false negative.

In a summary, the false positive mainly comes from hash collisions and inconsistency of multidimensional attributes. Hash collisions are an inherent property of hash functions and related research (e.g., reducing false positive from the hash function perspective) is beyond the scope of this paper. We mainly focus on the latter, i.e., inconsistency problem, by proposing a simple bit-based verification scheme as shown in the following Section 3.3. Instead, the false negative depends upon the given value of the parameter R and we observe that the false negatives can be significantly reduced by probing not only the hashed bits, but also their adjacent ones within limited steps. The main reason is that locality sensitive hashing functions can hash proximate items into the same or neighboring bits.

Checking operations on both the hashed bits and neighbors can cover more approximate members, thus improving query accuracy. A side effect of performing the checking on more bits is the possible increments of false positives. Therefore, it is important to determine the number of checked bits when taking into account both false positives and false negatives. An intuitive idea is that if the false negatives count more, the number should be relatively large. Otherwise, the number should be relatively small.

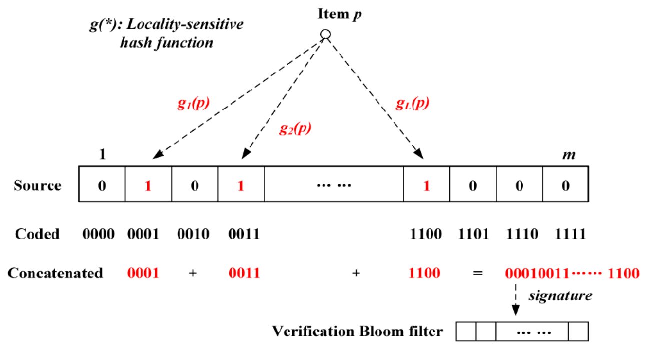


Fig. 4. Bit-based verification to decrease false positives.

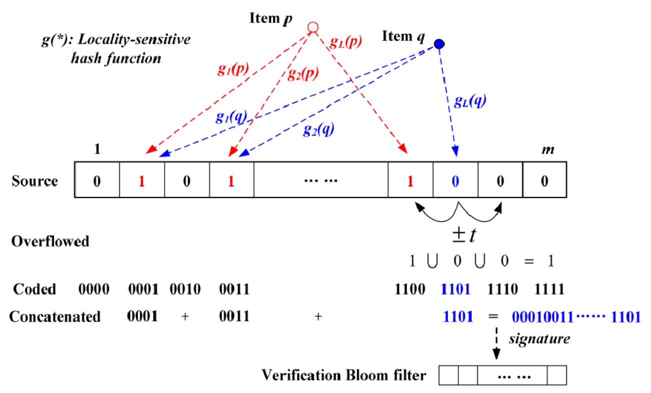
3.3 Bit-Based Verification Scheme for Decreasing FP

We present a bit-based verification scheme to decrease false positives in LSBF as shown in Fig. 4. The verification scheme needs to code each bit of LSBF according to its position with a binary number, related to the array size m. The code size can be taken as $\lceil \log_2 m \rceil$. Fig. 4 shows an example of the 4-bit code for each position. When an item p of a data set S is inserted into LSBF, the codes at the hashed "1" bits are utilized. We thus obtain the codes of item p as "0001," "0011," "$\cdots$" and "1100," which stay at the hashed positions and are further concatenated to produce the $L \cdot \lceil \log_2 m \rceil$-bit signature of item p, i.e., "$signature(p) = 00010011\ldots1{,}100$." The signature of item p is hashed into a *verification Bloom filter* that is a standard Bloom filter and consists of $m' = \lceil \frac{k \cdot n}{\ln 2} \rceil$ bits to maintain the signature of stored n items in the data set S. According to the conclusion of Bloom filter size in [24] to obtain the minimum false positive probability $(1/2)^k$ ($\approx (0.6185)^{m'/n}$), k should take $\frac{m'}{n} \ln 2$ where k is the number of hash functions, m' is the array size of the standard Bloom filter and n is the number of inserted items. The required space for the verification Bloom filter is very small in practice, i.e., $m' = \lceil \frac{k \cdot n}{\ln 2} \rceil$ bits. An AMQ hence needs to first examine the LSBF array for approximate membership and then verification Bloom filter for consistency. The verification scheme can satisfy the requirements for query accuracy for most real-world applications.

Given such verification scheme, an approximate membership query of item q needs to check the LSBF array and possibly the verification Bloom filter. First, we hash q into the LSBF array by using locality-sensitive hashing functions to check whether all hashed bits are "1." If this is the case, we proceed with the concatenation operation on the codes positioned at all hashed "1" bits to generate the signature of item q. We further hash the signature into the verification Bloom filter to check the exact-matching presence of the signature of item q. If it is a member in the verification Bloom filter, we say that item q is an approximate member.

3.4 Active Overflowed Scheme for Decreasing FN

Although performing above verification scheme decreases the potential false positive, it cannot help to mitigate false negatives that come from probabilistic property of locality sensitive hashing functions when hashing proximate items. To minimize false negatives, we propose an *active overflowed* scheme to identify proximate items that are unfortunately hashed into the neighboring bits.



g(*): Locality-sensitive hash function
Item p
Item q
g1(p)
g2(p)
gL(p)
g1(q)
g2(q)
gL(q)
1
m
Source
0 1 0 1 …… 1 0 0 0
Overflowed
±t
1 ∪ 0 ∪ 0 = 1
Coded 0000 0001 0010 0011 1100 1101 1110 1111
Concatenated 0001 + 0011 + 1101 = 00010011…… 1101
signature
Verification Bloom filter

Fig. 5. Active overflowed scheme to decrease false negatives.

In essence, the active overflowed scheme belongs to multiprobe LSH scheme, which exploits the fact that LSH can hash two close-by items into the same or adjacent buckets. Typical variants include multiprobe similarity search [2], prior knowledge-based multiprobe LSH [25] and bounded LSH [26]. If two close-by items p and q are not hashed into the same bucket, it is highly possible for their hashing to "close-by" buckets. Therefore, performing the multiprobe on neighboring buckets is likely to improve query quality. The active overflowed scheme will concatenate binary-bits representing queried result positions, which avoids potential complex union and rank operations.

Fig. 5 illustrates the scheme that adds extra overflowed check on adjacent t (here $t = 1$) bits, i.e., $g_L(q) - 1$ and $g_L(q) + 1$, when the hashed bit is "0." If any of adjacent $\pm t$ neighbors is "1," we conjecture that the hashed bit at $g_L(q)$ should be "1" to further generate its signature. In Fig. 5, the left neighbor, $g_L(q) - 1$, is "1" and thus the code "1101" will be used to represent the hashed bit to carry out the same verification of item q as in the basic scheme.

The active overflowed scheme can decrease false negatives by exploiting the locality-sensitive distribution when hashing proximate items. However, it possibly leads to increased false positives since we need to check neighboring bits of hashed bit in the verification process. In the active overflowed scheme, false positives and negatives depend upon two main factors. The first factor is related to the condition to take the approach, that is, the number of g functions with hashed "0" bits to proceed with the overflowed checking. The larger the number of such g functions that we choose, the smaller the false negatives that we can achieve. However, it will inevitably lead to the increment of false positives since we check more bits. The second factor is the t value, that is, the number of adjacent neighbors to check. Like the first one, a larger t will benefit false negative but throttle against false positives. In addition, Fig. 5 shows that the left neighbor of hashed bit is "1," which then indicates the approximate membership of queried item q for further verification. However, if the right neighbor of hashed bit is "1," we will also have to take into account it as one representative to execute the following verification by using "1110." Thus, we use 2 bits (left and right neighbors) to represent item q and when one is verified to be approximate member, the item q is accepted, which meanwhile may introduce false positives.

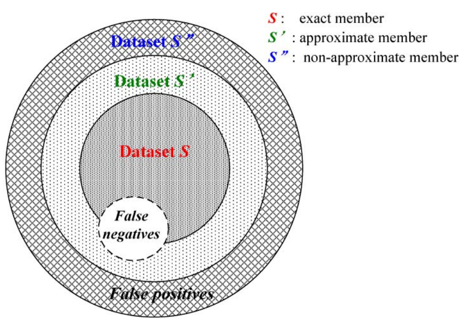


Fig. 6. Illustration of false positive and false negative occurrence.

Therefore, we should balance false positives and negatives when using the active overflowed scheme, depending on the accuracy requirement in different real-world applications. We can set a balanced false rate $f_{balance}$, i.e.,

$$f_{balance} = \lambda f^{+} + (1 - \lambda) f^{-},$$

where $\lambda \in (0, 1)$ is a trade-off parameter, f^{+} and f^{-}, respectively, indicate false rates of positives and negatives. In practice, if the ω in (1) is reasonably large, most proximate items can be hashed to the same bit [2]. Therefore, we restrict our attention to carry out the overflowed check for a single g function with "0" bit and $t = 1$.

4 Theoretical Analysis

We analyze characteristics of the proposed LSBF structure. First, we study the query quality by examining the probability of false positive and false negative. Then, we show the query time complexity and required space overhead.

In LSBF, L locality-sensitive hash functions are designed independently and can share the same hash collision probability for approximate items by satisfying Definition 3. Fig. 6 illustrates the occurrence of false positive and false negative in the LSBF structure where data sets S, S', and S'', respectively, represent, to the original items, the set of exact-matching members, approximate members and nonapproximate members but are wrongly considered as approximate members. False positives must occur for queried items in S'' while false negatives possibly occur in both S and S' due to the probabilistic hashing property of LSH.

4.1 False Positive Probability

The false positive probability in the LSBF structure mainly comes from the potential hash collisions in L LSH hash functions. The hash family of locality sensitive functions follows the s-stable distribution that is defined as the limits of normalized sums of i.i.d. variables. Typical s-stable distributions include Cauchy distribution where $s = 1$ and Gaussian (normal) distribution where $s = 2$.

We first examine the false positive of the basic LSBF (as in Fig. 2) for its hash collision probability and then the one using the bit-based verification scheme (as in Fig. 4).

Definition 4 (s-stable Distribution). *Given a distribution D, it is called s-stable if for n real numbers $\mu_1, \ldots, \mu_n$ and variables $X_1, \ldots, X_n$, which exhibit independent and identical distribution D, the variable $\sum_i \mu_i X_i$ has the same distribution as variable $(\sum_i |\mu_i|^s)^{1/s} X$, where variable X follows the distribution D.*

We further study the c-approximate false positive probability of the basic LSBF structure, which mainly comes from the potential hash collisions [27].

Theorem 1 (False Positive Probability of Basic LSBF). *For a queried item q, the basic LSBF uses $g_i(1 \leq i \leq L)$ locality-sensitive hash functions that follow the s-stable distribution to identify an approximate membership and its false positive probability is*

$$f^{+}_{basic} = e^{L \cdot \ln\left(1-\prod_{i=1}^{n}(1-q_i)\right)}, \tag{2}$$

where $q_i = Pr_{\cdot a,b}[h_{a,b}(p_i) = h_{a,b}(q)]$ and p_i is an item in a data set S with n items ($1 \leq i \leq n$).

Proof. A false positive occurs when a nonapproximate item has its hashed bits to be "1" for all L locality sensitive hash functions in the condition of hash collisions. The false positive probability is hence tightly associated with collision probability of LSH that follows the s-stable distribution.

Let $f_s(t)$ be the probability density function of s-stable distribution. According to the conclusion in [28], the probability that two items p_i and q collide for a LSH, q_i, is

$$Pr_{\cdot a,b}[h_{a,b}(p_i) = h_{a,b}(q)] = \int_0^{\omega} \frac{1}{\kappa} f_s\left(\frac{t}{\kappa}\right)\left(1 - \frac{t}{\omega}\right)dt, \tag{3}$$

where $\kappa = \|p_i - q\|_s$, vector a is drawn from an s-stable distribution and vector b is uniformly drawn from $[0, \omega)$. Note that q_i is no less than P_1 if $\kappa \leq R$ and no bigger than P_2 if $\kappa > cR$ according to Definition 3.

The false positive for the query of item q means for each LSH hash function, it must collide with an item in set S. We use q_i to denote the collision probability to item p_i. Thus, the probability is $(1 - q_i)$ that q does not collide with item p_i for the first LSH hash function, which implies q does not collide with any one with the probability $\prod_{i=1}^{n}(1 - q_i)$. As a result, for the first LSH function, the collision probability is $1 - \prod_{i=1}^{n}(1 - q_i)$. Since false positive of q infers L LSH hash collisions, its probability is $(1 - \prod_{i=1}^{n}(1 - q_i))^L$, that is,

$$f^{+}_{basic} = \left(1 - \prod_{i=1}^{n}(1 - q_i)\right)^L = e^{L \cdot \ln\left(1-\prod_{i=1}^{n}(1-q_i)\right)}.$$

□

The improved LSBF as shown in Section 3.3 uses an extra verification Bloom filter to reduce false positives. The Bloom filter, storing the hashed verification values of n items, functions as a double-check of approximate items. In other words, item q is regarded as an approximate item when both L LSH hash functions return "1" and the Bloom filter shows a positive answer. Thus, the false positive for the improved LSBF appears only if it occurs in both the basic LSBF and verification Bloom filter, involving two independent hashing processes. Recall that the basic LSBF and verification Bloom filter, respectively, employ LSH and uniform hash functions as shown in Fig. 4. The false positive in verification Bloom filter can achieve the minimum about $(0.6185)^{m'/n}$. Therefore, we have the following false positive probability of LSBF when it takes the simple but efficient verification scheme in the additional Bloom filter.

Corollary 2 (False Positive Probability of LSBF). *LSBF uses $g_i(1 \leq i \leq L)$ locality-sensitive hash functions and a bit-based verification Bloom filter. For a queried item q, its false positive probability is*

$$\begin{aligned} f^{+}_{improved} &= f^{+}_{basic} * (1/2)^k \\ &\approx f^{+}_{basic} * (0.6185)^{m'/n} \\ &= e^{L \cdot \ln(1-\prod_{i=1}^{n}(1-q_i))} * (0.6185)^{m'/n}, \end{aligned}$$

while assuming that LSH functions follow the s-stable distribution to identify an approximate membership.

4.2 False Negative Probability

A false negative occurs for the queried item q if $\|p, q\| \leq R$ and $[h(p) \neq h(q)]$ (named a *hash miss*) for one LSH function when p is a member of the queried data set S. According to Definition 3, the probability for the hash collision of p, q is not less than P_1.

Theorem 3 (False Negative Probability of Basic LSBF). *Given a queried item q, the basic LSBF uses L locality-sensitive hash functions to examine its approximate membership and the false negative probability is*

$$f^{-}_{basic} < 1 - P_1^L.$$

Proof. The false negative appears if at least one of hit bits is "0," which is different from false positives that occur if all hit bits are "1." Note that the probability of hash collision should be no less than P_1 in one LSH hash function for items p and q. Thus, the probability is no less than P_1^L for their hash collision in L LSH hash functions. Therefore, the probability is less than $1 - P_1^L$ that q is not regarded to be the approximate member of p. Due to the hashing probabilistic property in LSH, even if p and q do not collide for one LSH hash function, the hit bit for q can likely be set to "1" by others. In this case, the false negative will not occur. Hence, the false negative probability must be smaller than $1 - P_1^L$, i.e., $f^{-}_{basic} < 1 - P_1^L$. □

The P_1 value describes the acceptable quality of queried results. Given a data set under a specified metric space, we can determine P_1 value based on predefined parameters [13].

Observation 1 (P_1 Principle). *We define $\rho = \frac{\ln 1/P_1}{\ln 1/P_2}$ and LSH can guarantee to run with time complexity $O(n^{\rho})$ sublinearly to n if $\rho \leq 1/c$.*

The P_1 value depends on special applications using different metric spaces.

Case 1 (Hamming Distance). *$\forall c, R$ in a d-dimensional Hamming space, the probability $Pr_{\cdot a,b}[h_{a,b}(p) = h_{a,b}(q)]$ is equal to the rate when the coordinates of p and q collide. Thus, $P_1 = 1 - \frac{R}{d}$, $P_2 = 1 - \frac{cR}{d}$ and $\rho = 1/c$ [11].*

Case 2 (ℓ_s Distance). *The picking probability of hashed points into segments with ω follows the Gaussian distribution. For ℓ_2 euclidean distance and s-stable distribution and $s \in [0, 2)$, we have $\rho < 1/c$ [27].*

Case 3 (Arccos). *Given points $p, q \in \mathbb{R}^d$ and their distance measured by cosine angle of $\cos(\frac{p \cdot q}{\|p\| \cdot \|q\|})$, the collision probability of p and q is $1 - \arccos(\frac{p \cdot q}{\|p\| \cdot \|q\|})/\pi$ when we take into account the hash function as the space partition [29].*

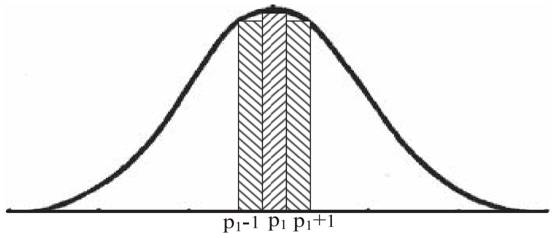


Fig. 7. Neighboring slots for approximate query of a point q.

We give the false negative probability of the improved LSBF by using the active overflowed scheme below. The active approach is similar to and further improves upon the multiprobe LSH [2] to probe neighboring bits when taking $t = 1$. Given a queried item q, assume that its approximate item in a database S is p_1. Let the hash collision probability be q_1 for item p_1 and q that can be computed by (3). Note that $q_1 \geq P_1$. From the conclusion in the multiprobe LSH, the projection difference of $(a \cdot p_1 + b) - (a \cdot q + b)$ follows a normal distribution. When ω is large enough, item q can fall into the same slot as item p_1, or its left and right neighboring slots (denoted by $\delta \in \{0, -1, +1\}$ slot, respectively) as shown in Fig. 7.

From [2], we know that item q falls into one of two neighboring slots of item p_1 with the approximate probability $\frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{v^2}{2\sigma^2}} dv \approx e^{-Cx(\delta)^2}$ for a locality-sensitive hash function, where the variance σ^2 is proportional to the distance $\|p_1 - q\|^2$, the constant C depends upon the distance $\|p_1 - q\|$ and

$$x(\delta) = \begin{cases} (a \cdot q + b) - \lfloor \frac{a \cdot q + b}{\omega} \rfloor \times \omega, & \text{if } \delta = -1, \\ \omega - (a \cdot q + b) + \lfloor \frac{a \cdot q + b}{\omega} \rfloor \times \omega, & \text{if } \delta = +1. \end{cases}$$

Therefore, the probability that q falls into the neighboring slots (i.e., $\delta \in \{-1, +1\}$) of item p_1 becomes $\Psi(x) \approx \sum_{\delta=-1,+1} e^{-C \cdot x(\delta)^2}$. In our active overflowed scheme to reduce false negatives, we need to check neighboring 2 bits for a single hash miss by setting $t = 1$. The single miss will not lead to negative answer for the proximity of item p and q if any neighboring bit is 1. Thus, the probability is approximately $(1 - q_1) \sum_{\delta=-1,+1} e^{-C \cdot x_i(\delta)^2}$ for one hash function and the hash collision of q only occurs in either left or right neighboring slot of item p_1.

Theorem 4 (False Negative Probability of LSBF). *Given that the improved LSBF uses L locality-sensitive hash functions and the active overflowed scheme probes adjacent $t = 1$ neighbors for a single hash miss of a queried item q, the false negative probability of LSBF is*

$$f^{-}_{improved} \approx 1 - q_1^L - L \cdot q_1^{L-1}(1 - q_1) \sum_{\delta=-1,+1} e^{-C \cdot x_i(\delta)^2}. \quad (5)$$

Proof. A false negative occurs for queried item q when there are two or more hash misses or a single miss but without falling into p's neighboring slots for $t = 1$ among L hashing functions. Thus, the false negative probability for the improved LSBF is $1 - Pr.(no\ miss) - Pr.(a\ miss\ but\ within\ neighboring\ slots)$. From the proof of Theorem 3, we have $Pr.(no\ miss) = q_1^L$ and $Pr.(a\ miss) = \binom{L}{1} \cdot q_1^{L-1}(1 - q_1)$. Therefore, based on the fact that the differentiated projection of item p and q follows Gaussian distribution, we can obtain

$$\begin{aligned} &Pr.(a\ miss\ but\ within\ neighboring\ slots) \\ &= Pr.(a\ miss)\Psi(x) \\ &\approx \binom{L}{1} \cdot q_1^{L-1}(1 - q_1) \sum_{\delta=-1,+1} e^{-C \cdot x_i(\delta)^2} \\ &= L \cdot q_1^{L-1}(1 - q_1) \sum_{\delta=-1,+1} e^{-C \cdot x_i(\delta)^2}. \end{aligned}$$

□

4.3 Query Time and Space Overheads

We give the query time and space overheads for LSBF using the verification scheme to improve query accuracy for a data set S containing n items. LSBF is composed of two components, the basic LSBF and verification Bloom filter.

Theorem 5 (Query Time and Space Overheads). *The query time and space overheads for LSBF in response to approximate query services are, respectively, $O(dL) + O(k)$ complexity and $O(dn/\omega) + O(n \cdot k/ln2)$ bits.*

Proof. Suppose the approximate query is processed sequentially for each locality-sensitive function. The query operation needs to first check d dimensions of a queried item, each of which requires to verify L LSH functions to examine the approximate membership in the basic LSBF. The first step requires $O(dL)$ computational complexity. To improve query accuracy, LSBF uses a verification Bloom filter and the check on it requires another $O(k)$ complexity due to the computation of k hash functions in the Bloom filter.

The storage space in LSBF consumes 2-bit vectors, respectively, in the basic LSBF and verification Bloom filter. The space overhead in the basic LSBF depends upon, besides the number of items n and dimensions d, the amount of slots divided by ω. Given a certain length of projected line, the larger the ω is, the smaller the number of slots will be, each of which is correlated with a "0/1" bit. Thus, the basic LSBF requires $O(dn/\omega)$-bit space. In addition, the verification Bloom filter constructs an m'-bit vector as stated in Section 3.3, which is $O(n \cdot k/ln2)$. □

5 Performance Evaluation

This section examines the performance of LSBF by using real-world traces with high dimensions. We compare its performance with other state-of-the-art work to support AMQ. To answer a query of q, LSBF first verifies its approximate membership presence in the basic LSBF and verification Bloom filter. LSBF then makes use of an active overflowed scheme to reduce FN only if a single hit bit is "0." Specifically, we present the implementation details, including environment setting, used data traces, generated query requests, and compared methods. We further demonstrate the real performance results in terms of query latency, query accuracy, I/O access cost, and space overhead.

5.1 Implementation Details

5.1.1 Environment Setting

We have implemented the proposed LSBF structure in a distributed environment that consists of a cluster with 30 nodes, each of which runs on the Linux and is equipped with dual-AMD processors and 2 GB memory and connected with a high-speed network. We carry out experiments 20 times to validate the results.

5.1.2 Traces and Queries

In order to comprehensively evaluate the performance of LSBF structure, we use real-world traces, including *MSN* [30] and *Forest CoverType* [31], to examine and compare LSBF performance with other comparable work. The following real-world data traces have been used in our evaluation.

- ***MSN Trace***. The *MSN* trace [32] maintains metadata information and correlated users within a 6-hour period and has been divided into 10-minute intervals. This trace contains 1.25 million files and records 3.3 million "READ" and 1.17 million "WRITE" operations. The queried objects using this trace are the files that exhibit multidimensional attributes, including access time, amounts of READ, amounts of WRITE, operational sequence IDs and file size within an examined interval.
- ***Forest CoverType Trace***. The *Forest CoverType* data set contains 581,012 data points and each has 54D attributes. Specifically, these 54D attributes include 10 quantitative variables, 4 binary wilderness areas and 40 binary soil type variables. All data records for forest cover types exhibit locality distribution. The detailed information of the data set can be found at the website [31]. We further carry out a simple preprocessing by extracting 11D attributes to be used in our experiments due to their significant differentiation on data points.

Query requests are generated from the attribute space of above typical traces and are randomly selected by considering the 1,000 uniform and 1,000 Zipfian distributions, respectively. We set the Zipfian parameter H to be 0.85. We select these 2,000 query requests to constitute the query set and examine the query accuracy and latency.

5.1.3 Compared Methods

AMQ in fact can be interpreted as top-1 Nearest Neighbor (NN) query by first identifying the closest neighbor to the queried point and then measure their distance. If the distance is smaller than the metric R, we say the queried point is an approximate member to data set S. Therefore, we compare LSBF with query methods for top-1 NN searching.

We have implemented LSBF in real distributed systems to facilitate comparisons with the baseline approach and other state-of-the-art work, including SmartStore [33] and LSB-tree [34] for AMQ. In fairness, SmartStore will first search top-1 nearest neighbor point and then compute the distance between the queried point and this top-1 nearest neighbor. If the distance is smaller than R, SmartStore will return a positive answer to the approximate membership query. We choose LSB-tree for comparisons because LSB-tree is the most recent work that can obtain high-quality AMQ results. We compare LSBF with state-of-the-art methods for approximate data query, including:

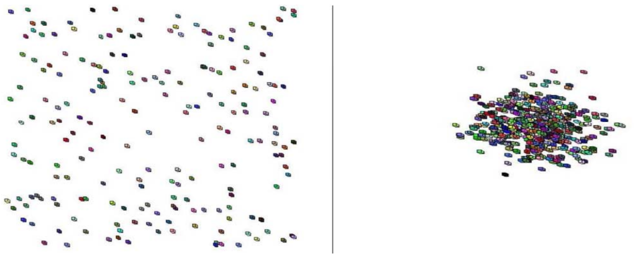


Fig. 8. Two typical data distributions. (a) Uniform. (b) Gauss.

- **Baseline**. The baseline approach utilizes the basic brute-force retrieval to identify the closest point in the data set and determine the approximate membership by computing the distance between the queried point and its closest neighbor.
- **SmartStore**. The SmartStore system uses information retrieval tool, latent semantic indexing (LSI), to semantically aggregate associated file metadata into the same or adjacent groups in an R-tree [35] to support approximate query service in Exabyte-scale file systems.
- **LSB-tree**. The LSB-tree uses LSH to provide approximate query service in the high-dimensional NN search by converting each point to the one in an m-dimensional space, where Z-order method is used to produce associated values that are indexed by a conventional B-tree.

5.1.4 Parameter Selection

The performance of LSBF structure is tightly associated with the parameter settings. One of the key parameters is the metric R to regulate the measure of approximate membership. The LSH-based structures can work well if R is roughly equivalent to the distance between the queried point q and its exact NN. Unfortunately, identifying an optimal R value is a nontrivial task due to the uncertainties and probabilistic properties of LSH. Too large or too small R values possibly result in bad query results [26]. Even worse, the optimal R value in fact does not exist at all and sometimes exhibits case-by-case fashion, i.e., a good R working for some cases may be bad for others [34]. In particular, a heuristic approach [36] attempts to find a "magic" radius predetermined by the system. Unfortunately, it is difficult to choose an optimal R value that can measure the nearest neighbor distances for all queries. For instance, as shown in Fig. 8, if a query falls in the uniform distribution, the distance to its NN must be much larger if the query falls into the Gauss distribution. Therefore, performing the selection of R value has to face with the dilemma between query efficiency and quality guarantee of approximation.

In order to obtain R values for carrying out experiments, we use the sampling method that is proposed in the LSH statement [27] and practical applications [34]. We further define "*proximity measure* $\chi = \|p_1^* - q\|/\|p_1 - q\|$" to evaluate the top-1 query quality for queried point q, where p_1^* and p_1, respectively, represent the actual and searched nearest

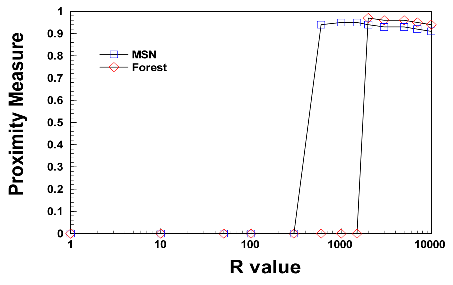


Fig. 9. Parameter R tuning for two typical traces by examining the weighted average value of proximity measure χ.

neighbors of point q by computing their euclidean distance. Fig. 9 shows the weighted average values of proximity measure ratios of typical traces. We observe that when the R value is too small, the ratio is close to 0 due to no retrieved results. With the increments of R value, we can obtain relatively satisfying ratios. However, if the R value is considerably large, the ratio may become worse because approximate points are too many and this could deteriorate their difference. Due to the space limitation, we ignore the details of discussing the distribution of measured points [26], [34]. We determine the R values to be 600 and 2,000, respectively, for *MSN* and *Forest* traces. We further initialize the size of LSBF structure to be $m = 10$ KB and corresponding verification Bloom filter to be $m' = 2$ KB. In addition, we use $L = 7$ LSH to support AMQ with $\omega = 0.85, M = 10$ and use eight hash functions, i.e., $k = 8$, in the verification Bloom filter in the function of MD5. We select $\lambda = 0.3$ for the computation of false rates since false negatives may generally lead to more severe effects than false positives. For the active overflowed approach, we set $t = 1$.

5.2 Performance Analysis

5.2.1 Query Latency

Figs. 10 and 11, respectively, show the query latency in the *MSN* and *Forest* traces. The Baseline uses linear searching on the entire data set and thus has the longest query latency, which can potentially cause inaccurate query results due to stale information of delayed updates. SmartStore leverages semantic grouping to reduce searching scope into several groups. However, it requires precomputation on matrix-based correlation analysis. Both the LSB-tree and LSBF make use of LSH to determine the approximate memberships. The main difference is that LSB-tree needs to index a B-tree with $O(\log n) - scale$ complexity after the hashing computation, while LSBF examines the verification Bloom filters with $O(1)$ complexity. In addition, we also observe that the query latency in the *MSN* trace is relatively smaller than that in the *Forest* trace, although the former contains more points than the latter. We conjecture that the main reason is that the *Forest* trace has much higher dimensions, thus requiring longer checking time.

5.2.2 Query Accuracy

We examine the query accuracy of LSBF and compared methods by using the metric of average "*proximity measure*" in the *MSN* and *Forest* traces as shown in Figs. 12 and 13. Note that false results here take into account both false positives and false negatives. We observe that the LSBF structure can

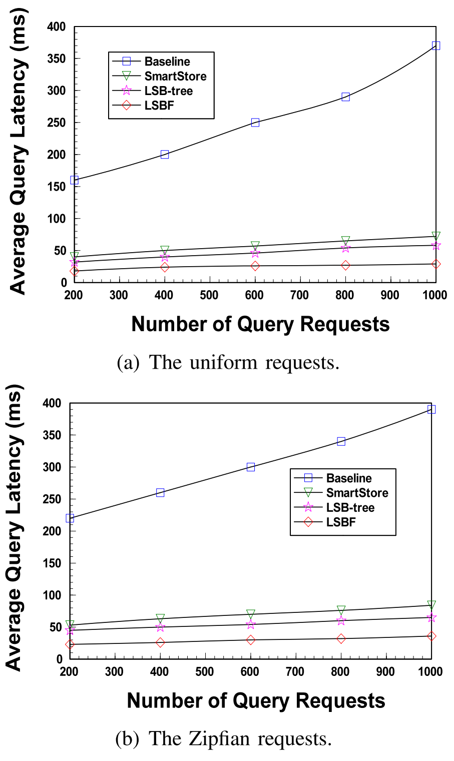


Fig. 10. Query latency with the *MSN* trace.

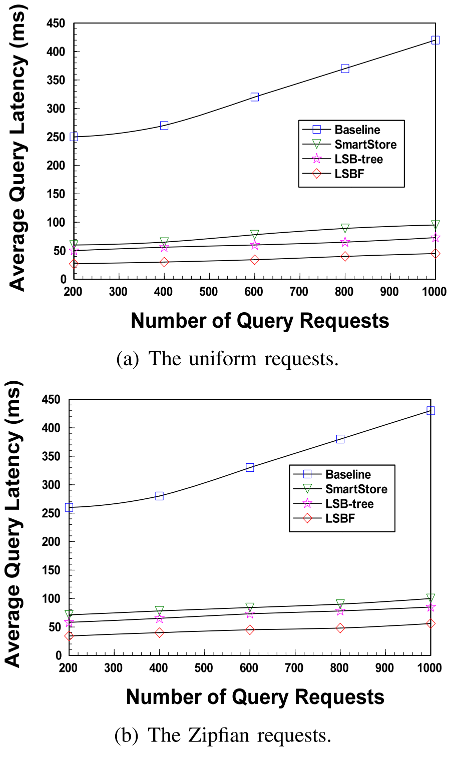
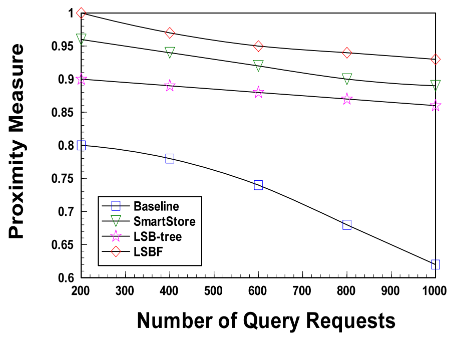


Fig. 11. Query latency with the *Forest* trace.

obtain accuracy advantage over SmartStore, LSB-tree, and baseline approaches. The main reason is that LSBF uses simple but efficient verification mechanism to guarantee query quality without compromising query performance. As illustrated in Fig. 12, SmartStore has higher accuracy than the LSB-tree in the *MSN* trace since the semantic grouping in the


(a) The uniform requests.

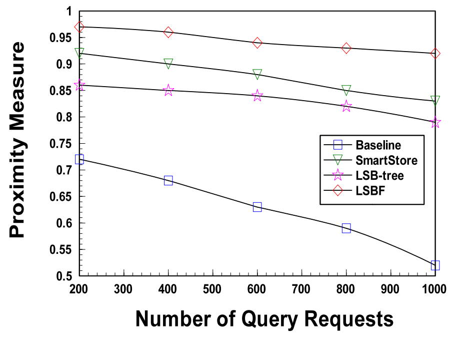
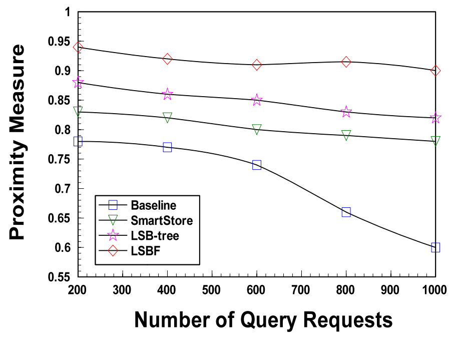

(b) The Zipfian requests.

Fig. 12. Query accuracy with the *MSN* trace.


(a) The uniform requests.

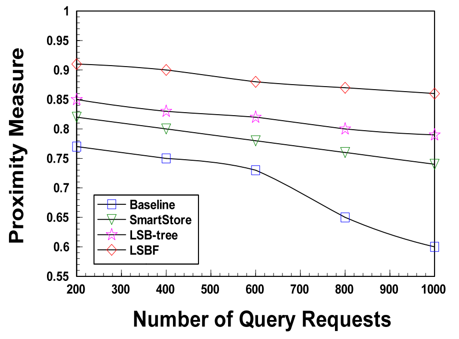
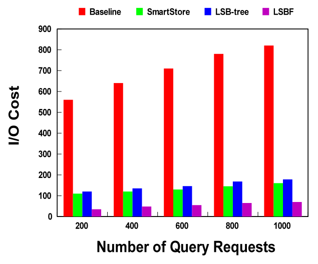

(b) The Zipfian requests.

Fig. 13. Query accuracy with the *Forest* trace.

former helps identify approximate members within one or a small number of groups. However, this is not the case for the *Forest* trace, as shown in Fig. 13, since SmartStore has to spend much more time to compute the correlation matrix that comes from the high-dimensional *Forest* trace, thus leading to stale results. In addition, the uniform distribution receives higher query accuracy than the Zipfian because the items in the latter are naturally closer and it is difficult to clearly identify them. The baseline approach uses brute-force search to obtain queried points but has the worst performance. Its slow response to update information in multiple nodes greatly degrades the query accuracy.


(a) The uniform requests.

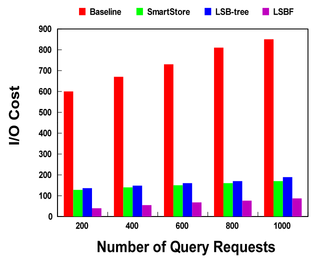
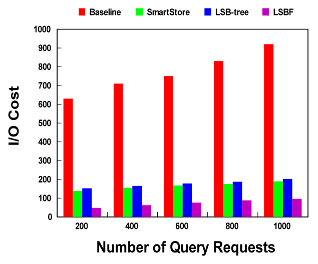

(b) The Zipfian requests.

Fig. 14. I/O costs with the *MSN* trace.


(a) The uniform requests.

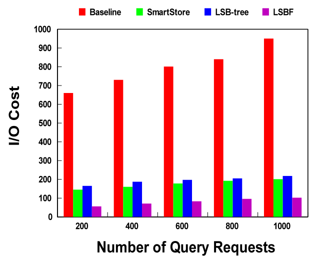

(b) The Zipfian requests.

Fig. 15. I/O costs with the *Forest* trace.

TABLE 1
Normalized Storage Space for Real-World Traces

	SmartStore	LSB-tree	LSBF
MSN	1.0	0.68	0.15
Forest	1.0	0.42	0.09

5.2.3 *I/O Access*

We count the I/O access times to evaluate query efficiency. Figs. 14 and 15 illustrate the I/O costs for approximate membership queries. The counted I/O access includes the visits on high-speed memory and low-speed disk. The baseline exhibits the largest number of accesses due to its checking on the entire data set. Performing the index on a B-tree makes LSB-tree to produce a little more visits than SmartStore. LSBF requires the smallest I/O visits since it only needs to verify limited bits by using hash-based computation and its space savings allow most visits to be in the memory.

5.2.4 *Space Overhead*

Table 1 shows the compared space overhead that is normalized to the space size of SmartStore. LSBF shows its advantage of significant space savings over other designs due to its bit-form storage for its basic LSBF and verification Bloom filter. We can thus put the entire LSBF structure into high-speed memory to obtain quick query responses. We also observe that more space consumption is intended for SmartStore than LSB-tree since the former needs to maintain complete information for high-dimensional items into an R-tree, while the latter only keeps the simple Z-order codes in a B-tree.

5.3 Performance Summary

Compared with state-of-the-art schemes, such as LSB-tree and SmartStore, LSBF can obtain significant performance improvements, in terms of approximate query accuracy and latency, storage space overhead and I/O costs, when we use traces from typical real-world applications for testing. Specifically, compared with LSB-tree and SmartStore, LSBF improves the query accuracy on average by 11.36 and 17.92 percent, decreases the query latency by 36.28 and 47.51 percent, saves almost 90 percent and 77 percent space overheads, and reduces 62.8 and 70.3 percent I/O costs, respectively.

6 Real-World Application Scenarios of LSBF

We can exploit the LSBF structure in many network applications because it has the following properties:

- First, LSBF makes use of LSH functions to efficiently support AMQ, which is very helpful to quickly respond query requests from users to return approximate answers. The conventional exact-matching, however, may require prohibitively brute-force searching costs and cause a long time delay, which limits its potential applications. For instance, in the decision support systems, exact answers are not required while approximate but early feedback can help to identify interesting regions.
- Second, LSBF can tolerate certain occasional input errors that often occur in practice due to human behaviors. It allows us to obtain approximate results from the uncertain and inaccurate inputs by leveraging the locality-aware hashing computation.
- Third, LSBF can provide the results close to the input requests, even if we have little knowledge of stored data. A client can send queries with vague description of multidimensional attributes of queried items and receive useful answers from LSBF since it is a structure exploiting the locality of multidimensional attributes.

The proposed LSBF structure can be efficiently used in many real-world applications due to its properties of input mistake tolerance, fast query response, assistance to similar query and system performance improvement. We demonstrate these properties in the following examples.

Example 1 (Mistake Tolerance). LSBF can tolerate certain occasional input mistakes. For example, "I would like to find a file created in June 2009, unfortunately, I mistakenly type the input as July 2009".

Example 2 (Fast Query Response). For example, in the file archiving application, it is normal for a user to forget the exact filename of previously stored files (e.g., emails/images). In order to avoid linearly brute-force searching on all files, LSBF can definitely help to find similar files by carrying out a multiattribute query, such as "$Time_{created} \approx July, 2006$" and "$Size \approx 20$ KB".

Example 3 (Similar Query). For example, AMQ can be very valuable for the indexing service of similar images on the web [2]. Performing the AMQ over LSBF can identify "computer"-related images that are represented by multidimensional attribute vectors.

Example 4 (System Performance Improvement). We can apply LSBF to improve approximate query performance, reduce searching space and provide good compatibility by using simple I/O interfaces. In fact, LSBF is orthogonal to existing schemes, such as Magellan [37] and SmartStore [33].

7 Related Work

AMQ has received lots of attentions due to their wide applications. Locality Sensitive Hashing introduced by Indyk and Motwani in [11] has been successfully applied in approximate queries of vector and string spaces. We can refer to [13] as a detailed survey. Existing variants include distance-based hashing [38], multiprobe LSH [2], and bounded LSH [26]. Distance-based hashing [38] extends conventional LSH into arbitrary distance measures by taking statistical observation from sample data. Multiprobe LSH [2] checks the hashed buckets more than once to support high-dimensional similarity search and improve indexing accuracy based on statistic analysis. Most of existing LSH-based designs have to consume a large storage space to maintain multiple hash tables to improve the accuracy of approximate queries. Bounded LSH [26] maps nonuniformly distributed data points into load-balanced hash buckets to contain approximately equal number of points, thus obtaining space savings.

Another research branch aims to extend space-efficient Bloom filters to support approximate queries with acceptable false rates. A new multiset data structure [7] serves as a substitute for standard Bloom filters for constant lookup time, smaller space usage, and succinct hash function encodings. It stores an approximation set S' to a set S such that $S \subseteq S'$ and guarantees that any element not in S belongs to S' with a probability at most ϵ. Distance-sensitive Bloom filters [12] allow Bloom filters to answer queries of the form, "Is x close to an element of S in a given metric?". The distance-sensitive Bloom filters are essentially standard partitioned Bloom filters where the random hash functions are replaced by distance sensitive hash functions. Beyond Bloom filters [3] can represent concurrent state machines by supporting "false positive," "false negative," and "don't know" query responses. These filters specifically use d-left hashing to provide membership checking in a dynamic set.

False positives and false negatives become an essential topic of Bloom filters due to hash collisions and environmental settings [39]. Weighted Bloom filters [40] incorporate the information on the query frequencies and the membership likelihood of elements into its optimal design. Data popularity conscious Bloom filters [41] study the problem of minimizing the false positive probability of Bloom filters by adapting the number of hashes used for each object to its popularity in set and membership queries. A partitioned hashing method [42] tries to reduce the false positive rate of Bloom filters by tailoring the hash functions for each item to set far fewer Bloom filter bits than standard Bloom filters. Incremental Bloom filters [23] consider the problem of minimizing the memory requirements in cases where the number of elements in a set is unknown in advance but only with its the distribution or moment information. Retouched Bloom filters [43] present a combination of false positive and negative rates by allowing the removal of certain false positives at the cost of producing random false negatives. Other details can be referred to the survey of Bloom filters [24].

The above existing work motivates our design of LSBF that makes further improvements upon them. LSBF is a novel structure to use LSH to map proximate items into the same or adjacent bits in a Bloom filter, thus supporting AMQ and obtaining space savings without compromising query performance.

8 Conclusion

In this paper, we proposed a novel structure, called LSBF, to support AMQ. LSBF is essentially a space-efficient Bloom filter but replaces original uniform hashing functions with LSH functions that can faithfully maintain the proximity of hashed items. A simple replacement with LSH in Bloom filters, however, may cause unsatisfactory answers due to false positives and negatives. To decrease false positives, we presented a verification scheme by using an extra small-size Bloom filter to keep consistency of multidimensional attributes of items. The proposed active overflowed scheme can further help decrease false negatives by probing very limited neighbors of hashed bits. We showed theoretical analysis on LSBF, e.g., its FP, FN, and compact storage space overhead. Extensive experiments in a distributed environment verify that the LSBF design can be used in real-world applications for its space efficiency and responsive query answers.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant 61173043; National Basic Research 973 Program of China under Grant 2011CB302301; NSFC under Grant 61025008, 60703046; HK RGC PolyU 5314/10E; Fundamental Research Funds for the central universities, HUST, under grant 2010MS043; National High Technology Research and Development 863 Program of China under Grant 2009AA01A401 and 2009AA01A402; the Program for Changjiang Scholars and Innovative Research Team in University under Grant IRT-0725. The authors greatly appreciate anonymous reviewers for constructive comments.

References

[1] L. Carter, R. Floyd, J. Gill, G. Markowsky, and M. Wegman, "Exact and Approximate Membership Testers," *Proc. 10th Ann. ACM Symp. Theory of Computing (STOC '78),* pp. 59-65, 1978.

[2] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-Probe LSH: Efficient Indexing for High-Dimensional Similarity Search," *Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB '07),* pp. 950-961, 2007.

[3] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "Beyond Bloom Filters: From Approximate Membership Checks to Approximate State Machines," *Proc. ACM SIGCOMM,* 2006.

[4] Y. Zhu and H. Jiang, "False Rate Analysis of Bloom Filter Replicas in Distributed Systems," *Proc. Int'l Conf. Parallel Processing (ICPP '06),* pp. 255-262, 2006.

[5] W. Chang Feng, D.D. Kandlur, D. Saha, and K.G. Shin, "Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness," *Proc. IEEE INFOCOM,* 2001.

[6] F.M. Cuenca-Acuna, C. Peery, R.P. Martin, and T.D. Nguyen, "PlantP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities," *Proc. IEEE 12th Int'l Symp. High Performance Distributed Computing,* 2003.

[7] A. Pagh, R. Pagh, and S. Rao, "An Optimal Bloom Filter Replacement," *Proc. 16th Ann. ACM-SIAM Symp. Discrete Algorithms (SODA '05),* pp. 823-829, 2005.

[8] S. Dharmapurikar, P. Krishnamurthy, and D.E. Taylor, "Longest Prefix Matching Using Bloom Filters," *Proc. ACM SIGCOMM,* pp. 201-212, 2003.

[9] A. Broder and M. Mitzenmacher, "Using Multiple Hash Functions to Improve IP Lookups," *Proc. IEEE INFOCOM,* pp. 1454-1463, 2001.

[10] F. Baboescu and G. Varghese, "Scalable Packet Classification," *IEEE/ACM Trans. Networking,* vol. 13, no. 1, pp. 2-14, Feb. 2005.

[11] P. Indyk and R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," *Proc. 13th Ann. ACM Symp. Theory of Computing (STOC '98),* pp. 604-613, 1998.

[12] A. Kirsch and M. Mitzenmacher, "Distance-Sensitive Bloom Filters," *Proc. Eighth Workshop Algorithm Eng. and Experiments (ALENEX),* 2006.

[13] A. Andoni and P. Indyk, "Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions," *Comm. ACM,* vol. 51, no. 1, pp. 117-122, 2008.

[14] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Trans. Networking,* vol. 8, no. 3, pp. 281-293, June 2000.

[15] M. Mitzenmacher, "Compressed Bloom Filters," *IEEE/ACM Trans. Networking,* vol. 10, no. 5, pp. 604-612, Oct. 2002.

[16] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, "Scalable and Adaptive Metadata Management in Ultra Large-scale File Systems," *Proc. 28th Int'l Conf. Distributed Computing Systems (ICDCS '08),* pp. 403-410, 2008.

[17] A. Kumar, J.J. Xu, J. Wang, O. Spatschek, and L.E. Li, "Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement," *Proc. IEEE INFOCOM,* pp. 1762-1773, 2004.

[18] C. Saar and M. Yossi, "Spectral Bloom Filters," *Proc. ACM SIGMOD Int'l Conf. Management data* , pp. 241-252, 2003.

[19] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and Network Application of Dynamic Bloom Filters," *Proc. IEEE INFOCOM,* 2006.

[20] B. Xiao and Y. Hua, "Using Parallel Bloom Filters for Multi-Attribute Representation on Network Services," *IEEE Trans. Parallel and Distributed Systems,* vol. 21, no. 1, pp. 20-32, Jan. 2010.

[21] H. Song, F. Hao, M. Kodialam, and T.V. Lakshman, "IPv6 Lookups Using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards," *Proc. IEEE INFOCOM,* 2009.

[22] F. Hao, M. Kodialam, T.V. Lakshman, and H. Song, "Fast Multiset Membership Testing Using Combinatorial Bloom Filters," *Proc. IEEE INFOCOM,* 2009.

[23] F. Hao, M. Kodialam, and T.V. Lakshman, "Incremental Bloom Filters," *Proc. IEEE INFOCOM,* pp. 1741-1749, 2008.

[24] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Math.,* vol. 1, pp. 485-509, 2005.

[25] A. Joly and O. Buisson, "A Posteriori Multi-Probe Locality Sensitive Hashing," *Proc. ACM Multimedia,* 2008.

[26] Y. Hua, B. Xiao, D. Feng, and B. Yu, "Bounded LSH for Similarity Search in Peer-to-Peer File Systems," *Proc. Int'l Conf. Parallel Processing,* pp. 644-651, 2008.

[27] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni, "Locality-Sensitive Hashing Scheme Based on p-Stable Distributions," *Proc. Ann. Symp. Computational Geometry,* pp. 253-262, 2004.

[28] A. Andoni, M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni, "Locality-Sensitive Hashing Using Stable Distributions," *Nearest Neighbor Methods in Learning and Vision: Theory and Practice,* T. Darrell and P. Indyk and G. Shakhnarovich, eds., MIT Press, 2006.

[29] M. Charikar, "Similarity Estimation Techniques from Rounding Algorithms," *Proc. 34th Ann. ACM Symp. Theory of Computing (STOC '02),* pp. 380-388, 2002.

[30] N. Agrawal, W. Bolosky, J. Douceur, and J. Lorch, "A Five-Year Study of File-System Metadata," *Proc. Fifth USENIX Conf. File and Storage Technologies (FAST),* 2007.

[31] The Forest CoverType data set, "UCI Machine Learning Repository," http://archive.ics.uci.edu/ml/data sets/Covertype, 2011.

[32] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of Storage Workload Traces from Production Windows Servers," *Proc. IEEE Int'l Symp. Workload Characterization (IISWC),* 2008.

[33] Y. Hua, H. Jiang, Y. Zhu, D. Feng, and L. Tian, "SmartStore: A New Metadata Organization Paradigm with Semantic-Awareness for Next-Generation File Systems," *Proc. ACM/IEEE Supercomputing Conf. (SC),* 2009.

[34] Y. Tao, K. Yi, C. Sheng, and P. Kalnis, "Quality and Efficiency in High-Dimensional Nearest Neighbor Search," *Proc. 35th SIGMOD Int'l Conf. Management of Data (SIGMOD '09),* 2009.

[35] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '84),* pp. 47-57, 1984.

[36] A. Gionis, P. Indyk, and R. Motwani, "Similarity Search in High Dimensions via Hashing," *Proc. 25th Int'l Conf. Very Large Data Bases (VLDB '99),* pp. 518-529, 1999.

[37] A. Leung, I. Adams, and E.L. Miller, "Magellan: A Searchable Metadata Architecture for Large-Scale File Systems," Technical Report UCSC-SSRC-09-07, Univ. of California, Nov. 2009.

[38] V. Athitsos, M. Potamias, P. Papapetrou, and G. Kollios, "Nearest Neighbor Retrieval Using Distance-Based Hashing," *Proc. IEEE 24th Int'l Conf. Data Eng. (ICDE),* 2008.

[39] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian, "Supporting Scalable and Adaptive Metadata Management in Ultra Large-Scale File Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 22, no. 4, pp. 580-593, Apr. 2011.

[40] J. Bruck, J. Gao, and A. Jiang, "Weighted Bloom Filter," *Proc. IEEE Int'l Symp. Information Theory,* pp. 2304-2308, 2006.

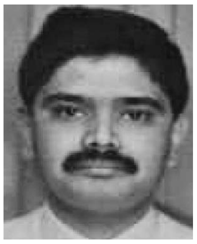
[41] M. Zhong, P. Lu, K. Shen, and J. Seiferas, "Optimizing Data Popularity Conscious Bloom Filters," *Proc. 27th ACM Symp. Principles of Distributed Computing (PODC '08),* 2008.

[42] F. Hao, M. Kodialam, and T. Lakshman, "Building High Accuracy Bloom Filters Using Partitioned Hashing," *Proc. ACM SIGMETRICS Int'l Conf. Measurement and Modeling of Computer Systems (SIGMETRICS '07),* pp. 277-288, 2007.

[43] B. Donnet, B. Baynat, and T. Friedman, "Retouched Bloom Filters: Allowing Networked Applications to Trade off Selected False Positives Against False Negatives," *Proc. ACM CoNEXT Conf.,* 2006.

Yu Hua received the BE and PhD degrees in computer science from the Wuhan University, China, in 2001 and 2005, respectively. He joined the Department of Computing at the Hong Kong Polytechnic University as a research assistant in 2006. Now he is an associate professor in the School of Computer Science and Technology at the Huazhong University of Science and Technology, China. His research interests include computer architecture, cloud computing, network storage, and cyber-physical systems. He has more than 30 papers to his credit in major journals and international conferences including *IEEE Transactions on Computers (TC)*, *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, USENIX ATC, INFOCOM, SC, ICDCS, ICPP, and HiPC. He has been on the program committees of multiple international conferences. He is a member of the IEEE and USENIX.

Bin Xiao received the BSc and MSc degrees in electronics engineering from Fudan University, China, in 1997 and 2000, respectively, and PhD degree in computer science from University of Texas at Dallas in 2003. Now he is an associate professor in the Department of Computing of the Hong Kong Polytechnic University, Hong Kong. His research interests include distributed computing systems, data management, secured communication networks, focusing on wireless sensor networks, and RFID systems. Currently, he is the associate editor of the *International Journal of Parallel, Emergent and Distributed Systems*. He is a member of the IEEE Computer Society and a senior member of the IEEE.



Bharadwaj Veeravalli received the BSc degree in physics, from Madurai-Kamaraj University, India in 1987, the master's degree in electrical communication engineering from Indian Institute of Science, Bangalore, India in 1991 and the PhD degree from the Department of Aerospace Engineering, Indian Institute of Science, Bangalore, India in 1994. He did his postdoctoral research in the Department of Computer Science, Concordia University, Montreal, Canada in 1996. He is currently with the Department of Electrical and Computer Engineering, Communications and Information Engineering (CIE) division, at The National University of Singapore, as a tenured associate professor. Currently, he is serving the editorial board for *IEEE Transactions on SMC-A*, and *Multimedia Tools & Applications*, as an associate editor. His main stream research interests include, multiprocessor systems, cloud/cluster/grid computing, scheduling in parallel and distributed systems, bioinformatics & computational biology, and multimedia computing. He is one of the earliest researchers in the field of divisible load theory (DLT). He has published more than 85 papers in high-quality International Journals and several papers in international conferences. He had successfully secured several externally funded projects. He has served the editorial board of *IEEE Transactions on Computers*. He is a senior member of the IEEE and IEEE Computer Society.

Dan Feng received the BE, ME, and PhD degrees in computer science and technology from the Huazhong University of Science and Technology (HUST), China, in 1991, 1994, and 1997, respectively. She is a professor and vice dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 80 publications to her credit in journals and international conferences, including *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, JCST, USENIX ATC, FAST, ICDCS, HPDC, SC, ICS and ICPP. She is a member of the IEEE.