ISSN No. 0976-5697

Volume 2, No. 5, Sept-Oct 2011

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Optimization and Applications of Dynamic Bloom Filters

M.M.Siva Krishna*,V.Bala Sankar
[1]MCA (M.Tech (cse), [2]Asst. Professor in Dept. of CSE
Sri Sai Aditya Institute of Science and Technology, Surampalem
East Godavari (dt), Andhra Pradesh,India
*[1]sivakrishna.munaga@gmail.com, [2]balasankar.v@gmail.com

Abstract: Bloom Filters (BF) are space-efficient data structures that allow membership queries from a set. The Bloom Filters and its variants just focus on how to represent a static set and decrease the false probability to a sufficiently low level. By look into the applications based on the Bloom Filters, we reveal that dynamic datasets are more common and important than static sets. But the existing variants of the Bloom Filters cannot support dynamic data sets well. To address this issue Dynamic Bloom Filters (DBF) has been proposed as a method to implement Bloom Filters in a scalable environment, i.e. where the final size of a dataset is not known in advance. DBF seems to be a logical addition to BF for a scalable environment - just before the false positive (FP) rate of a particular BF starts growing fast, we simply switch to a new filter and store the old one.DBF handles inserts and lookups. We present multi-dimension dynamic bloom filters (MDDBF) to support concise representation and approximate membership queries of dynamic sets in multiple attribute dimensions, and study the false positive probability and union algebra operations. We also explore the optimization approach and three network applications of bloom filters, namely bloom joins, informed search, and global index implementation.

Key words: BF, FP, DBF, MDDBF

I. INTRODUCTION

Bloom Filters were suggested in 1970 but have recently gained increased momentum. The major variations of bloom filters include compressed bloom filters [1], counting bloom filters [2], distance-sensitive bloom filters [3], bloom filters with two hash functions [4], space-code bloom filters [5], and spectral bloom filters [6]. Compressed bloom filters can improve performance in terms of bandwidth saving when bloom filters are passed on as messages. Counter bloom filters deal mainly with the element deletion operation of bloom filters. Distance-sensitive bloom filters, using locality-sensitive hash functions, try to answer queries of the form, “Is x close to an element of S?”. Bloom filters with two hash functions use a standard technique in hashing to simplify the implementation of bloom filters significantly.

Space-code bloom filters and spectral bloom filters are approximate representation of a multi set, which allows for querying, “How many occurrences of x are there in set M?”. Both bloom filters and their variations are suitable for representing static sets whose size can be estimated before design and deployment.

Although the SBF and its variations have found suitable applications in different fields, the following three obstacles still lack suitable and practical solutions:

a. As the actual size of a data set increases, its corresponding bloom filter should scale well in order to avoid too much deviation between the actual false positive probability and the predefined threshold. In order to solve this problem, we introduce dynamic bloom filters (DBF) to support concise representation and approximate membership queries of dynamic sets.
b. How to represent dynamic sets to support queries based on multiple attributes? We propose multi-dimension dynamic bloom filters (MDDBF) to support concise representation and approximate membership queries of dynamic set in multiple attribute dimensions.
c. How to implement an efficient and scalable informed search protocol in unstructured P2P networks? We propose a framework of informed search based on bloom filters, and evaluate the positive impact of bloom filter through simulation.

The basic idea of dynamic bloom filters is to represent a dynamic set with a dynamic s×m bit matrix that consists of s standard bloom filters. We prove that DBF can control the false positive probability at a low level if DBF dynamically adjusts the number of standard bloom filters used according to the actual number of elements that belong to the given set. Furthermore, the space complexity is also acceptable if the estimation of the maximum size of the dynamic set does not deviate too much from the actual one. The most related work is split bloom filters [7] which use a constant s×m bit matrix to represent a set, where s is a constant and must be pre-defined according to the estimation of the maximum value of set size. However, split bloom filters waste too much storage space and bandwidth before the actual size of the given set reaches (m × ln 2)/k. Furthermore, a split bloom filter needs to be reconstructed when the actual size of the given set exceeds the estimation value. On the contrary, DBF naturally overcomes these disadvantages.

II. CONCISE REPRESENTATION AND MEMBERSHIP QUERIES OF STATIC SETS

A. *Standard Bloom Filters:*

A Bloom filter for representing a set X={x_1, . . . ,xn} of n items is described by a vector of m bits, initially all set to 0. A Bloom filter uses k independent hash functions h_1, . . . ,hk to map each item of X to a random number over a range {1, . . .,m} [8],[9] uniformly. For each item x of X, we define its Bloom filter address as *Bfaddress(x)*, consisting of h_i(x) for $1\leq i \leq k$, and the bits belonging to *Bfaddress(x)* are set to 1 when inserting x. Once the set X is represented as a Bloom filter, to judge whether an element x belongs to X, one just needs to check whether all hi(x) bits are set to 1. If so, then x is a member of X. Otherwise, we assume that x is not a member of X. It is clear that a Bloom filter may yield a false positive due to hash collisions, for which it suggests that an element x is in X even though it is not. The reason is that all indexed bits were previously set to 1 by other items [8].

The probability of a false positive for an element not in the set can be calculated in a straightforward fashion, given our assumption that hash functions are perfectly random. Let p be the probability that a random bit of the Bloom filter is 0, and let n be the number of items that have been added to the Bloom filters. Then, $p = (1\text{-}1/m)^{n*k} \approx e^{-n*k/m}$ as n*k bits are randomly selected, with probability 1/m in the process of adding each item. We use $f^{BF}_{m,k,n}$ to denote the false positive probability caused by the (n+1)th insertion, and we have the expression:

$$f^{BF}_{m,k,n} = (1\text{-}p)^k \approx (1\text{- } e^{-n*k/m})^k \qquad (1)$$

In the remainder of this paper, the false positive probability is also called the false match probability. We can calculate the filter size and number of hash functions given the false match probability and the set cardinality according to (1) from [8]. We know that the minimum value of $f^{BF}_{m,k,n}$ is $0{:}6185^{m/n}$ when k=(m/n) ln 2. In practice, of course, k must be an integer, and smaller k might be preferred since that would reduce the amount of computation required.

For a static set, it is possible to know the whole set in advance and design a perfect hash function to avoid hash collisions. In reality, an SBF is usually used to represent dynamic sets as well as static sets. Therefore, it is impossible to know the whole set and design k perfect hash functions in advance. On the other hand, different perfect hash functions used by an SBF may cause hash collisions. Thus, the perfect hash functions are not suitable for overcoming hash collisions in SBFs in theory, as well as practice.

On the other hand, a static set is typically not allowed to perform data addition and deletion operations once it is represented by an SBF. Thus, the bit vectors of the SBF will stay the same over time, and then, the SBF can correctly reflect the set. Therefore, the membership queries based on the SBF will not yield a false negative in this scenario. However, the SBF must commonly handle a dynamic set that is changing over time, with items being added and deleted.

In order to support the data deletion operation, an SBF hashes the item to be deleted and resets the corresponding bits to 0. It may, however, set a location to 0, which is also mapped by other items. In such a case, the SBF no longer correctly reflects the set and will produce false negative judgments with high probability. To address this problem, Fan et al. introduced counting Bloom filters (CBFs) [2]. Each entry in the CBF is not a single bit but rather a small counter that consists of several bits. When an item is added, the corresponding counters are incremented; when an item is deleted, the respective counters are decremented. The experimental results and mathematical analysis show that four bits for each counter is large enough to avoid overflows [2].

III. CONCISE REPRESENTATION AND MEMBERSHIP QUERIES OF DYNAMIC SET

DBF focuses on addressing dynamic sets with changing cardinality rather than static sets, which were addressed by the previous version. It should be noted that DBFs can support static sets. Throughout this paper, an SBF is called *active* only if its false match probability does not reach a designed upper bound; otherwise, it is called *full*. Let n_r be the number of items accommodated by an SBF. The n_r is equal to the capacity *c* for a full SBF and less than *c* for an active SBF. In the rest of this paper, we use SBF to imply counting Bloom filters for the sake of supporting the item deletion operation.

A. *Overview of Dynamic Bloom Filters:*

A DBF consists of *s* homogeneous SBFs. The initial value of *s* is 1, and the initial SBF is active. The DBF only inserts items of a set into the active SBF, and appends a new SBF as an active SBF when the previous active SBF becomes full. The first step to implement a DBF is initializing the following parameters: the upper bound on false match probability of the DBF, the largest value of *s*, the upper bound on false match probability of the SBF, the filter size *m* of the SBF, the capacity c of the SBF, and number of hash functions *k* of the SBF. As we will discuss further on in this paper, the approaches used to initialize these parameters re not identical in different scenarios.

Alg 1. Insert (x)

```
Require: x is not null
1: ActiveBF← GetActiveStandardBF()
2: if ActiveBF is null then
3:     ActiveBF← CreateStandardBF(m, k)
4:     Add ActiveBF to this dynamic Bloom filter.
5:     s ←s+1
6: for i = 1 to k do
7: ActiveBF[hash_i(x)] ←ActiveBF[hash_i(x)]+1
8: ActiveBF.nr← ActiveBF.nr+1
GetActiveStandardBF()
1: for j = 1 to s do
2:     if StandardBF_j.nr < c then
3:             Return StandardBF_j
4: Return null
```

Given a dynamic set X with n items, we will first show how a DBF is represented through a series of item insertion operations. Algorithm 1 contains the details regarding the process of the item insertion operation. It is clear that the DBF should first discover an active SBF when inserting an item x of X. If there are no active SBFs, the DBF creates a new SBF as an active SBF and increments s by one. The DBF inserts x into the active SBF and increments nr by one for the active SBF. If X does not decrease after deployment, only the last SBF of the DBF will be active, whereas the other SBFs are full. Otherwise, these full SBFs may become active if some items are removed from the set X.

It is convenient to represent X as a DBF by invoking Alg 1 repeatedly. After achieving the DBF, we can answer any set membership queries based on the DBF instead of X. The detailed process is illustrated in Alg 2, which uses an item *x* as input. If all the $hash_j(x)$ counters are set to a nonzero value for $1 \leq j \leq k$ in the first SBF, then the item *x* is a member of X. Otherwise, the DBF checks its second SBF, and so on. In summary, x is not a member of X if it is not found in all SBFs, and is a member of X if it is found in any SBF of the DBF.

Alg 2. Query (x)

```
Require: x is not null
1: for $i = 1$ to $s$ do
2:     $counter \leftarrow 0$
3:     for $j = 1$ to $k$ do
4:         if $StandardBF_i[hash_j(x)] = 0$ then
5:             break
6:         else
7:             $counter \leftarrow counter + 1$
8:     if $counter = k$ then
9:         Return true
10: Return false
```

If an item x is removed from X, the corresponding DBF must execute Algorithm 3 with x as the input in order to reflect X as consistently as possible. First of all, the DBF must identify the SBF in which all the $hash_j(x)$ counters are set to a nonzero for $1 \le i \le k$. If no SBF exists that satisfies the constraint in the DBF, the item deletion operation will be rejected since x does not belong to X. If there is only one SBF satisfying the constraint, the counters $hash_j(x)$ for $1 \le j \le k$ are decremented by one. If there are multiple SBFs satisfying the constraint, then x may appear to be in multiple SBFs of the DBF. Thus, it is impossible for the DBF to know which the right one is. If the DBF persists in removing membership information of x from it, the wrong SBF may perform the item deletion operation with given probability. The wrong item deletion operation destroys the DBF and leads to, at most, k potential false negatives. To avoid producing false negatives, the membership information of such items is kept by the DBF, but removed from X.

Alg 3. Delete (x)

```
Require: x is not null
1: $index \leftarrow null$
2: $counter \leftarrow 0$
3: for $i = 1$ to $s$ do
4:     if BF[i].Query(x) then
5:         $index \leftarrow i$
6:         $counter \leftarrow counter + 1$
7:     if $counter > 1$ then
8:         break
9: if $counter = 1$ then
10:  for $i = 1$ to $k$ do
11:     $BF[index][hash_i(x)] \leftarrow BF[index][hash_i(x)]-1$
12:     $BF[index].nr \leftarrow BF[index].nr-1$
13:     Merge()
14:     Return true
15: else
16:     Return false
Merge()
1: for $j = 1$ to $s$ do
2:   if $StandardBFj{:}n < c$ then
3:     for $k = j+1$ to $s$ do
4:       if $StandardBF_j.n_r + StandardBF_k.n_r < c$ then
5:         $StandardBF_j \leftarrow StandardBF_j \cup StandardBF_k$
6:         $StandardBF_j.n_r +\leftarrow StandardBF_k.n_r$
7:         Clear $StandardBF_k$ from the dynamic Bloom filter.
8: Break
```

Furthermore, two active SBFs should be replaced by the union of them if the addition of their nr is not greater than the capacity c of one SBF. The union operation of counting Bloom filters is similar to that of standard Bloom filters, which performs the addition operation between counter vectors instead of the logical or operation between bit vectors. Note that there is at most one pair of SBFs which satisfy the constraint of union operation after an item is removed from the DBF.

The average time complexity of adding an item x to an SBF and a DBF is the same: $O(k)$, where k is the number of hash functions used by them. The average time complexities of membership queries for SBF and DBF are $O(k)$ and $O(k+s)$, respectively. The average time complexities of a member deletion for SBF and DBF are $O(k)$ and $O(k+s)$, respectively.

IV. CONCISE REPRESENTATION AND MEMBERSHIP QUERIES OF MULTI-ATTRIBUTE DYNAMIC SET

A. *Multi-Dimension Dynamic Bloom Filters:*

Standard and dynamic bloom filters just focus on representing sets consisted of single attribute objects, and supporting approximate membership queries based on a single attribute. In reality, it is common to describe and represent a given object using multiple attributes in many applications. In order to deal with this situation, we propose multi-dimension standard bloom filters (MDBF) and multi-dimension dynamic bloom filters (MDDBF). The basic idea is to represent sets consisted of multi-attribute objects from each attribute dimension using standard and dynamic bloom filters. In the following discussion, we first explain the details of adding objects with multi attribute to a MDDBF in Alg 4. Then add all the objects of a dynamic set A to the MDDBF according to Alg 4.

In order to represent multi-dimension information of a given object, we first obtain the DBF for each attribute dimension according to the attribute name from current MDDBF. Then, add the value of each attribute to the corresponding DBF by calling Alg 1. It is necessary to initialize every DBF for each attribute dimension before processing the first addition.

Alg 4. Insert (*element*)

```
Require: element with multi-attribute is not null
1: Get all attribute names of the element, and store them to a string array attributes
2: for $i = 0$ to $attributes.length$ do
3: $DynamicDBF \leftarrow GetDynamicDBF(attributes[i])$
4: if $DynamicDBF$ is null then
5:     $DynamicDBF \leftarrow CreateDynamicDBF(m, k)$
6:     SetDynamicBF(attribute[i], DynamicDBF)
7: DynamicDBF.Insert(element.GetValue(attribute[i]))
```

Once dynamic set A has been represented as an MDDBF, we check whether an element is a member of set A according to the MDDBF instead of the set A itself. We present the details of the algorithm of supporting membership queries based on the value of multi-attribute in Alg 4.

```
Alg 2. Query (element)
Require: element with multi-attribute is not null
1: Get all attribute names of element, and store them to a string array attributes
2: for $i = 0$ to attributes.length do
3: DynamicDBF ← GetDynamicDBF(attributes[i])
4: if DynamicDBF.Query (element.GetValue(attributes[i])) is false then
5:         Return false
6: Return true
```

The major process of Algorithm 4 is as follows. First, find the corresponding DBF for each attribute dimension of an element. Second, check whether the value of element for each attribute dimension is presented by corresponding DBF by invoking Alg 2. If the responses for all attribute dimensions are true, one can assume that element ∈ A with some false positive probability. Otherwise, one can be sure that element A.

The time complexity of adding an element to MDDBF is $O(l\times k)$, where l denotes the number of attribute dimensions used to describe the full information of a given object, and k denotes the number of hash functions used by dynamic bloom filters. The average time complexity of querying an object from a given MDDBF based on multi-attribute is $O(l \times k \times (s + 1)/2)$, where s is the number of standard bloom filters used by the DBF for each attribute dimension. Algorithms of multi-attribute set representation and membership queries are similar between MDBF and MDDBF, so these algorithms for MDBF are omitted here.

V. APPLICATIONS OF DYNAMIC BLOOM FILTERS

Bloom filters have a great potential for distributed protocols where systems need to share information about what data they have. A survey of network applications of bloom filters has been presented in [9]. Moreover, bloom filters as a better data structure has great potential for representing objects in memory [10], [11]. DBF is also suitable for various applications mentioned in those papers, and has some better characteristics than standard bloom filters.

In distributed applications, some peers own large amount of data while most of the nodes own a small amount of data. If we set relevant parameters of standard bloom filters according to the largest amount of data, it would result in huge waste of space and bandwidth. By adjusting the number of standard bloom filters used according to the actual number of data at each node, DBF can overcome this problem. Furthermore, DBF can tolerate the data increase without reconstructing a new bloom filter at each node. If a distributed application desires to distribute DBF of each peer among part of or all other peers, it also needs to keep the consistency among replications for each DBF. In reality, the data insertion just affects the active BF of DBF, and for keeping consistency it is enough to gossip the active BF instead of the whole DBF.

a. ***Bloom joins*****:** Bloom joins [12], [13] is a method for performing a fast join between two distributed data sets R_1 and R_2 based on a attribute a: R_1 in site 1 and R2 in site 2. The bloom joins includes the following steps. First, site 1 represents R_1 as a $BF(R_a)$ in the attribute dimension a and sends it to site 2. Second, site 2 sends tuples of R_2 with a match in $BF(R_a)$ to site 1, noted as R_{12}. Third, site 1 performs a join operation between R_1 and R_{12}, and produces the final result. The first transmission only sends a summarization of a projection of the tuples, and the second transmission usually contains a small fraction of the tuples. So this method is economical in network usage.

DBF is also suitable to perform single attribute distributed bloom joins between data sets as the number of tuples increases. Furthermore, MDDBF can be used to perform multi attribute distributed bloom joins between data sets as the number of tuples increases. In the following, we will give an example.

SELECT R.a, R.b, R.c, S.d, S.e FROM R, S
WHERE R.a = S.a and R.b=S.b

First, Site 1 represents data sets R as a $BF(R_{a,b})$ in the attribute dimensions a and b, and sends it to site 2. Second, site 2 sends tuples of data set S with a match in $BF(R_{a,b})$ to site 1, denoted as $R_{r,s}$. Third, at site 1, performs a join operation between R and $R_{r,s}$, and produces the final result.

b. ***Informed Routing*****:** The searching strategy in unstructured P2P systems is either blind search or informed search [14]. In a blind search such as iterative deepening [15] and random walker [16], no node has information about the location of the desired data. In an informed search [17], [18], each node keeps some information about the data location.

Bloom filters are an alternative method to implement informed resource routing for distributed applications, and many literatures have recently presented different approaches to utilize bloom filters for different scenarios [19], [20], [21], [22], [23]. The common assumption in those literatures is to represent local resource using bloom filters and gossip it to other peers according to different control mechanisms. Thus, each peer can possess individual bloom filters coming from related peers, then re-construct them according to the distance and/or direction between the local peer and other peers, and obtain a set of union results of individual bloom filters at each relative distance and/or relative direction.

A dynamic bloom filter is still suitable to support informed routing, and has more advantages than the standard one as the resource at each peer increases. Furthermore, a dynamic bloom filter is more suitable to support the necessary union operation than the standard one according to Theorem 5. As mentioned above, dynamic bloom filters, standard bloom filters, and their variations just represent objects and support approximate membership queries in a single attribute dimension. On the other hand, it often requires to route multi-attribute queries in reality. Both MDDBF and MDBF can satisfy this need. The former has the advantage over the latter as the resource at each peer increases, and supports the union operation better than the latter according to Theorem 7. Thus, DBF and its variations are better alternatives than standard bloom filters to implement informed routing in some scenarios.

c. ***Implementation of global index:*** We will refer to the globally replicated index as the global index, while the more detailed index that describes only the resources hosted locally by a peer will be denoted as the local index. Global index can be implemented in a number of ways. We define bloom filters in such a way that each peer summarizes the set of terms in its local index as a bloom filter. The cost of replicating the global index can be reduced by simply decreasing the gossiping rate; updating the global index with a new bloom filter requires constant time, regardless of the number of changes introduced. Furthermore, bloom filters can be compressed to achieve a single bit per word average ratio. Memory-constrained peers can also independently trade accuracy for storage by combining several filters into one.

When the global index has been established and propagated to the whole network, each peer uses a copy of global index hosted at local storage to find the desired peers and appropriate resources within one hop. In order to support queries that contain a set of queries based on different attribute dimensions, we can adopt MDDBF to summarize local content index and construct global content index by a periodic gossiping update operation.

VI. CONCLUSIONS

A bloom filter is a simple, space-efficient, randomized data structure for concisely representing a static data set in order to support approximate membership queries. As the actual size of the set increases continuously after deployment, a bloom filter should scale well in order to avoid too much deviation between the actual false positive probability and the predefined threshold. In order to deal with this problem, we present dynamic bloom filters to support concise representation and approximate membership queries of dynamic sets. It has been proved that dynamic bloom filters not only possess the advantage of standard bloom filters, but also have better features than standard bloom filters when dealing with dynamic sets. False positive probability of dynamic bloom filters can be controlled at a low level, and space complexity is also acceptable if the estimation of the threshold of the dynamic set does not deviate too much. In addition, we present multi-dimension dynamic bloom filters to support concise representation and approximate membership queries of dynamic sets from multiple attribute dimensions.

We have explored three kinds of representative applications of dynamic bloom filters: bloom joins, informed search, and implementation of global index. These applications also illustrate that dynamic bloom filters and their variations scale well and are practical for representing dynamic sets. Finally, we have simulated the informed search protocol based on bloom filters in unstructured P2P networks. Our simulation shows that informed search based on bloom filters can obtain high recall and success rate of query than the blind search protocol.

In future work, we will further enhance dynamic bloom filters in order to support the removal operation, and compare the space/time trade-off of both dynamic and standard bloom filters.

VII. REFERENCES

[1]. M Mitzenmacher. Compressed bloom Filters. IEEE/ACM Trans. networking, 10(5):604-612, 2002.

[2]. 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.

[3]. A. Kirsch and M. Mitzenmacher, "Distance-Sensitive Bloom Filters," Proc. Eighth Workshop Algorithm Eng. and Experiments (ALENEX '06), Jan. 2006.

[4]. A. Kirsch and M. Mitzenmacher, "Building a Better Bloom Filter," Technical Report tr-02-05.pdf, Dept. of Computer Science, Harvard Univ., Jan. 2006.

[5]. A. Kumar, J. Xu, J. Wang, O. Spatschek, and L. Li, "Space-Code Bloom Filter for Efficient Per-Flow Traffic Measurement," Proc. 23rd IEEE INFOCOM, pp. 1762-1773, Mar. 2004.

[6]. S. Cohen and Y. Matias, "Spectral Bloom Filters," Proc. 22nd ACM SIGMOD, pp. 241-252, June 2003.

[7]. M. Xiao, Y. Dai, and X. Li. Split bloom filters. Chinese Journal of Electronic, 32(2):241–245, 2004.

[8]. B. Bloom, "Space/Time Tradeoffs in Hash Coding with Allowable Errors," Comm. ACM, vol. 13, no. 7, pp. 422-426, 1970.

[9]. A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," Internet Math., vol. 1, no. 4, pp. 485-509, 2005.

[10]. C. Jin, W. Qian, and A. Zhou. Analysis and management of streaming data: A survey. Journal of Software, 15(8):1172–1181, 2004.

[11]. C. D. Peter and M. Panagiotis. Bloom filters in probabilistic verification. In Proc. the 5th International Conference on Formal Methods in Computer-Aided Design, pages 367–381, Austin, Texas, USA, November 2004.

[12]. L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In Proc. the 12^{th} International Conference on Very Large Data Bases (VLDB), pages 149-159, Kyoto, Jpn, August 1986.

[13]. Z. Li and K. A. Ross. Perf join: An alternative to two-way semi join and bloom join. In Proc. International Conference on Information and Knowledge Management, pages 137–144, Baltimore, MD, USA, November 1995.

[14]. X. Li and J. Wu. Searching techniques in peer-to-peer networks. In J. Wu, editor, Handbook of Theoretical and Algorithmic Aspects of Ad Hoc, Sensor, and Peer-to-Peer Networks. Auerbach, New York,USA, 2006.

[15]. B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. In Proc. the 22th IEEE International Conference on Distributed Computing, pages 5–14, Vienna, Austria, July 2002.

[16]. Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In Proc. the 16th ACM International Conference on Supercomputing, pages 84–95, Marina Del Rey, CA, United States, June 2002.

[17]. A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In Proc. the 22th International Conference on Distributed Computing, pages 23–32, Vienna, Austria, July 2002.

[18]. D. Tsoumakos and N. Roussopoulos. Adaptive probabilistic search in peer-to-peer networks. In Proc.

the 3th International Conference on Peer-to-Peer Computing, pages 102–109, Sweden, September 2003.

[19]. S. C. Rhea and J. Kubiatowicz. Probabilistic location and routing. In Proc. IEEE INFOCOM, pages 1248–1257, New York, NY, United States, June 2004.

[20]. T. D. Hodes, S. E. Czerwinski, and B. Y. Zhao. An architecture for secure wide-area service discovery. Wireless Networks, 8(2-3):213–230, 2002.

[21]. P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In Proc. ACM International Middleware Conference, pages 21–40, Rio de Janeiro, Brazil, June 2003.

[22]. D. Bauer, P. Hurley, R. Pletka, and M. Waldvogel. Bringing efficient advanced queries to distributed hash tables. In Proc. IEEE Conference on Local Computer Networks, pages 6–14, Tampa, FL, United States, November 2004.

[23]. K. Shanmugasundaram, H. Bronnimann, and N. Memon. Payload attribution via hierarchical bloom filters. In Proc. the 11th ACM Conference on Computer and Communications Security, pages 31–41, Washington, DC, United States, October 2004