IEICE TRANS. INF. & SYST., VOL.E94–D, NO.8 AUGUST 2011

PAPER

A Memory Efficient Result Cache Scheme for P2P DHT Based on Bloom Filters*

Takahiro ARIYOSHI[†], *Nonmember and* **Satoshi FUJITA**[†a)], *Member*

SUMMARY In this paper, we study the problem of efficient processing of conjunctive queries in Peer-to-Peer systems based on Distributed Hash Tables (P2P DHT, for short). The basic idea of our approach is to *cache* the search result for the queries submitted in the past, and to use them to improve the performance of succeeding query processing. More concretely, we propose to adopt Bloom filters as a concrete implementation of such a result cache rather than a list of items used in many conventional schemes. By taking such an approach, the cache size for each conjunctive query becomes as small as the size of each file index. The performance of the proposed scheme is evaluated by simulation. The result of simulation indicates that the proposed scheme is particularly effective when the size of available memory in each peer is bounded by a small value, and when the number of peers is 100, it reduces the amount of data transmissions of previous schemes by 75%.

key words: *peer-to-peer, Distributed Hash Table, conjunctive query, bloom filter*

1. Introduction

Efficient search of shared resources is a key issue in large distributed systems such as computational grid and Peer-to-Peer (P2P) systems. In the literature, a number of schemes have been proposed to solve such distributed file search problem, and among such schemes, information management based on the notion of Distributed Hash Table (DHT) has attracted considerable attentions in the past decade.

A P2P overlay based on DHT is commonly referred to as P2P DHT, and a number of concrete P2P DHT's have been proposed in the literature, such as Chord [7], CAN [4], Patry [6], and Tapestry [8]. As will be explained later, by using a mechanism provided by the original DHT, the search of files associated with a keyword could be efficiently realized in P2P DHT. However, if we want to find files associated with *multiple keywords*, the requester of such (conjunctive) queries must conduct a search for each keyword contained in the query, and must take an intersection of those results after collecting all of those search results from their corresponding peers. In addition, the amount of search result for a conjunction of keywords is generally much smaller than the search result for each keyword, which implies that we could significantly improve the performance of such a naive scheme with respect to the response time and the amount of message transmissions by designing an appropriate query processing scheme dedicated for conjunctive queries.

In this paper, we study the problem of efficient processing of conjunctive queries in P2P DHT. The basic idea of our approach is to *cache* the search result for conjunctions of the keywords processed in the past, and to use them to improve the performance of succeeding query processing. Such a cache is generally referred to as **result cache** [1] and has been widely used in the field of database systems. Here, the reader should note that there is a "freedom" in designing concrete result cache in actual environment, and we have to select an appropriate one according to the given environment such as CPU power, memory size, communication bandwidth, and so on. For example, a file index used in P2P DHT generally contains a unique ID of the file, name and the IP address of the file holder, and several keywords associated with the content of the file. If each index has a size of 1 KB, we should prepare 1 MB of memory to store 1000 indices. Thus, if we use a caching scheme which stores the search result for conjunctive queries in the form of an index list consisting of 1000 indices on average, it requires 1 GB to store 1000 lists (i.e., 1000 conjunctions) per peer. This value is apparently too large to ask to each user to dedicate space to the system, in such systems supported by the volunteers as in P2P.

This paper proposes a new implementation of result cache for P2P DHT with a limited capacity of the cache memory. In order to attain a sufficiently high hit rate to the result cache in such resource-restricted P2P DHTs, we need to bound the cache size required for each conjunction as small as possible. In conventional result cache schemes [3], [5], a list of indices is cached for each query (and for each sub-query), which consumes a large amount of storage, in the worst case. To overcome such drawback of conventional approaches, in this paper, we propose to use Bloom filters [2] as a concrete implementation of the result cache. By taking such an approach, the cache size for each query becomes as small as the size of each file index, although it (slightly) increases the frequency of the accesses to the DHT due to false positiveness of Bloom filters (details of the scheme will be given later). The performance of the proposed scheme is evaluated by simulation. The result of

Manuscript received November 4, 2010.
Manuscript revised February 25, 2011.

[†]The authors are with the Department of Information Engineering, Graduate School of Engineering, Hiroshima University, Higashihiroshima-shi, 739–8527 Japan.

*Earlier versions of this paper were presented at T. Ariyoshi and S. Fujita, "Efficient Processing of Queries with Multiple Keywords in P2P DHT with Limited Memory," In Proc. PDPTA 2010, pp.51–55, July 2010, and T. Ariyoshi and S. Fujita, "Efficient Processing of Conjunctive Queries in P2P DHTs Using Bloom Filter," In Proc. ISPA 2010, pp.458–464, September 2010.

a) E-mail: fujita@se.hiroshima-u.ac.jp

DOI: 10.1587/transinf.E94.D.1602

simulation indicates that the proposed scheme is particularly effective when the size of memory available in each peer for the result cache is bounded by a small value, and it reduces the amount of data transmissions of previous schemes by 36.6%.

The remainder of this paper is organized as follows. Section 2 describes a model of P2P DHT. Section 3 describes a basic caching scheme. The proposed method is given in Sect. 4. The simulation result is summarized in Sect. 5. Section 6 overviews related work. Finally, Sect. 7 concludes the paper with future work.

2. Model

Consider a P2P system consisting of a set of peers P. In the following, we assume that the set of peers is fixed, and peers in P have a homogeneous capability including the memory capacity and the network bandwidth. Peers in P can directly communicate with each other through an underlying network protocol. Each peer holds several files. Each file is attached a unique ID, and is associated with several **keywords** representing the content of the file.

The basic idea of DHT is described as follows. At first, consider a virtual coordinate space which will be partitioned into several subspaces such that each subspace is managed by each peer participating in P. Index of each file, i.e., the name and the IP address of the file holder, is stored at a *coordinate point* in the space which is calculated by applying an appropriate hash function to the file ID and/or a keyword associated with the file. Access to the stored information is realized by forwarding a message to the peer managing the point in the coordinate space. Such a message forwarding is realized by repeating message transmissions among nearby peers towards the direction in which the distance to the destination point is minimized, where nearby peers are those which manage subspaces adjacent in the whole coordinate space.

In this paper, we consider conjunctive queries consisting of several keywords. A conjunction of keywords is denoted as Q and R, and in the following, we will often identify a conjunctive query with a set of keywords contained in it. As was described above, in conventional P2P DHT, index of a file associated with a keyword is mapped to a point in the DHT, and is stored to the local storage of a peer who manages the point.

3. Basic Scheme

This section describes a basic scheme to process conjunctive queries in P2P DHT, which will be used in the proposed scheme as a building block. The basic idea of the scheme is to cache the search result for a given conjunctive query to the DHT including partial results obtained through the processing. More concretely, after obtaining a set of indices S matching a given query Q, it stores the following information to the DHT: 1) a string representing Q, 2) index set S, and 3) the name and the IP address of a peer holding S. The point in the DHT to which the above information is stored is calculated by applying a hash function to string Q, similar to conventional DHT.

An outline of the procedure is described as follows (a formal description the procedure will be given later): At first, a requester checks whether or not an index to the search result for Q is cached in the DHT. If it is cached, the requester simply acquires the search result from the point, and terminates the processing. Otherwise, it checks whether there exists a subset Q' of Q such that the search result for Q' is cached in the DHT, and if there exists such a subset, the requester calculates the result for Q by referring to the result for such subsets.

3.1 Notation

In order to formally describe the above idea, in the following, we introduce two functions Size and Sub. Function Size takes a conjunction R as a parameter, and returns the number of indices matching R if the result for R is cached in the DHT, and returns null otherwise. The reader should note that function Size is easily realized by allowing each peer to store the number of indices matching a conjunction to the DHT when it stores the result for the conjunction. Function Sub takes conjunction R as a parameter, and returns a set of sub-conjunctions of R, denoted by Ψ ($\subseteq 2^R$), satisfying the following two conditions:

- $\bigcup_{R' \in \Psi} R' = R$, and
- for any element $a \in R$, Ψ contains a singleton set $\{a\}$.

The first condition requests that R is covered by Sub-conjunctions contained in Sub(R). By this condition, it is guaranteed that we can calculate the search result for R by acquiring the set of indices matching R' for each $R' \in \Psi$ from the DHT (if any), and by taking an intersection of them. The second condition ensures that the above procedure works well even when the search result for conjunctions of length two or more is not cached in the DHT (note that if the set of indices matching conjunction $\{a\}$ is not stored in the DHT, it implies that there are no files matching keyword a). In the simulation given in Sect. 5, we fix Sub such that: 1) keywords in R are arranged in a lexicographical order, and 2) Sub(R) contains all conjunctions corresponding to the prefixes of the ordered sequence.

3.2 Algorithm

Let Q be a given query. By using the above two functions, the basic scheme can be described as follows:

Algorithm BASIC_CACHE

1. { When the result for entire Q is cached }
   If Size(Q) $\neq$ null, then acquire the search result for Q from the DHT, and terminate (dynamic update of files will be considered later).
2. { Calculation of set Ψ }

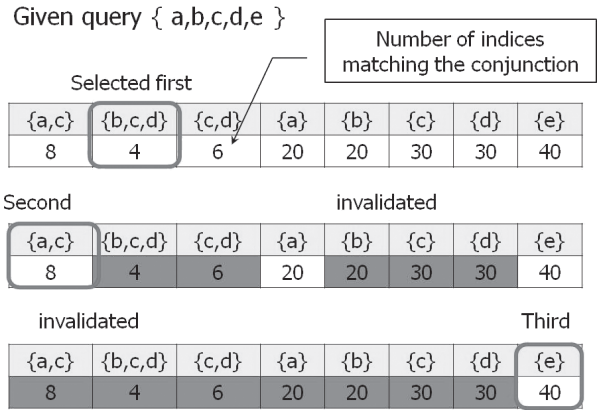


Fig. 1 Example 1.

Calculate Size(Q') for each $Q' \in$ Sub(Q) and construct a set of sub-conjunctions $\Psi(\subseteq$ Sub(Q)) consisting of Q''s whose search result is cached in the DHT. If $\bigcup_{Q' \in \Psi} Q' \neq Q$, then output $\emptyset$ as the search result, and terminate. Otherwise, proceed to the next step after initializing X and Ψ^* to $\emptyset$.

3. { Calculation of set Ψ^* }
   Repeat the following operations until X becomes Q: 1) Select Q' with a smallest Size in Ψ, and move it from Ψ to Ψ^*; 2) Update X as $X := X \cup Q'$; and 3) If Ψ contains Q'' such that $Q'' \subseteq X$, then remove it from Ψ.
4. { Calculation of the search result }
   For each $Q' \in \Psi^*$, acquire the search result for Q' from the DHT, and take their intersection. It then outputs the result of intersection as the search result for Q.
5. { Cache of the search result }
   Cache all partial results obtained during the processing of Q to the DHT, with their corresponding sub-conjunctions. Each peer participating in the DHT manages such cached information by the LRU (Least Recently Used) policy. That is, when the cache of a new conjunction causes an excess of the limit of the cache, then it expires a conjunction which is least recently accessed.

Note that Ψ^* obtained in Step 3 satisfies $\bigcup_{Q' \in \Psi^*} Q' = Q$ similar to Sub(Q) as long as the search result for Q is not empty. In addition, set Ψ^* is constructed in such a way that the number of indices matching each conjunction in Ψ^* is as small as possible. By evaluating keywords in Q in the order such that the amount of partial result is as small as possible, we can reduce the amount of data transmissions.

Example 1: Let $Q = \{a, b, c, d, e\}$ and Size(Q) = null (Step 1). Suppose that we have a set of sub-conjunctions Ψ, represented as follows (see Fig. 1 for illustration):

$$\Psi = \{\{a, c\}, \{b, c, d\}, \{c, d\}, \{a\}, \{b\}, \{c\}, \{d\}, \{e\}\},$$

and that the size of each conjunction is calculated as follows (Step 2):

$$8, 4, 6, 20, 20, 30, 30, 40.$$

Since $\{b, c, d\}$ is the smallest, we move it from Ψ to Ψ^*, and update X as $X := \{b, c, d\}$. As a result, Ψ contains several elements included by X. By removing them, Ψ is updated as $\Psi := \{\{a, c\}, \{a\}, \{e\}\}$. By repeating similar operations, we obtain Ψ^* as follows (Step 3):

$$\Psi^* = \{\{b, c, d\}, \{a, c\}, \{e\}\}.$$

We then cache the search result for $\{a, c, e\}$, $\{a, b, c, d\}$, $\{b, c, d, e\}$, and $\{a, b, c, d, e\}$, and terminate.

4. Proposed Method

BASIC_CACHE works effectively provided that the cache size of each peer is sufficiently large, since the search results will always be cached in the DHT, and the hit ratio to the result cache monotonically increases as increasing the elapsed time, unless expiration occurs. Unfortunately, each cache has a fixed size, and we cannot avoid expiration of cached data, which repeats the processing of the same query to increase the amount of transmitted data. In order to overcome such a problem, in this paper, we propose a new implementation of the result cache based on the notion of Bloom filter [2].

In the following, after providing a brief review of the Bloom filter, we describe the way of utilizing the Bloom filter for the query processing. We then describe how to apply the Bloom filter to the basic scheme described in the last section.

4.1 Bloom Filter

Let S be an arbitrary subset of a universe U. The basic idea of Bloom filter is to represent the containment of each element in U to S by using a bit array B of length m ($\ll |U|$), where we will refer to B as the Bloom filter concerned with subset S. Association of S to B is realized through k different hash functions $h_1, h_2, \ldots, h_k$ from U to $\{1, 2, \ldots, m\}$. More concretely, the record of an element $z \in S$ to B is realized by letting $B[h_j(z)] := 1$ for each $1 \leq j \leq k$, where all bits in B are initialized to zero. On the other hand, the containment of z to S is evaluated as follows:

$$\text{Judge } z \in S \text{ iff } B[h_j(z)] = 1 \text{ for all } 1 \leq j \leq k.$$

By definition, such a judge involves a false positiveness, i.e., it may misjudge "$y \in S$" even if it is not actually contained in S (note that misjudge in the reverse direction does not occur). It is known that the probability of such a misjudge depends on parameters m, k, and the cardinality of subset S [2]. In general usage of Bloom filter, such a probability is requested to be as small as possible, and it is generally selected such that the length of B is sufficiently large.

4.2 Bloom Filter as a Simple Index Filter

We can effectively use Bloom filters to reduce unnecessary data transmissions in P2P DHT (without using BASIC_CACHE) in the following manner. Recall that each file

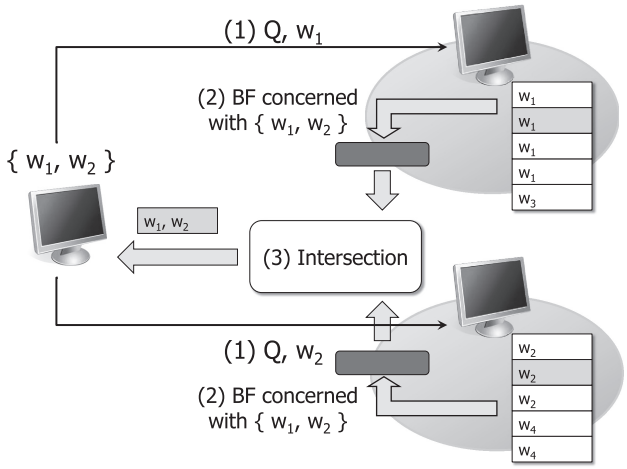


Fig. 2 Algorithm SIMPLE_FILTER.

held by each peer is attached a unique ID. Let S be a set of file indices which match a given conjunctive query Q. By preparing a Bloom filter B_Q corresponding to query Q, and by recording indices contained in S to B_Q, each peer can avoid to transmit indices which do not match Q, without conducting an actual processing of the conjunctive query (in the following, we will call such indices which match a certain keyword in Q but does not match the whole Q, as "unnecessary" indices). More concretely, by distributing bit sequence B_Q to all peers managing the indices concerned with keyword contained in Q, it can reduce the amount of indices transmitted by the peer when this query is issued in the next time. In other words, such a B_Q could be regarded as a "result cache" which represents the search result for Q by using a bit array of length m.

The above idea is realized by the following algorithm (see Fig. 2 for illustration):

Algorithm SIMPLE_FILTER

1. Let $Q = \{w_1, w_2, \ldots, w_x\}$ be a conjunctive query issued by a requester. The requester sends Q to peers managing file indices matching w_i for each $1 \leq i \leq x$ (arrow (1) in Fig. 2).
2. After receiving Q, a peer who manages indices matching w_i, conducts the following operation:
   a. Let S_i be the set of indices associated with keyword w_i.
   b. If it has a Bloom filter B_Q concerned with query Q, then it applies B_Q to S_i to eliminate indices which have no possibility of matching Q, and returns the result to the requester (item (2) in Fig. 2).
   c. Otherwise, if it has $B_{Q'}$ concerned with a conjunction $Q' \subset Q$, then it applies all such Bloom filters to S_i to obtain an index set. It then returns the result to the requester. Here, the reader should note that the false positiveness of Bloom filters does not violate the correctness of the overall algorithm; i.e., it never removes an index matching Q although it may pass several indices which do not match Q.
   d. Otherwise, it simply returns S_i to the requester.
3. After receiving all results, the requester takes an intersection of them, and obtains the search result S for query Q (item (3) in Fig. 2). If B_Q does not exist, then it generates B_Q from S, and multicasts it to all peers managing file indices concerned with a keyword contained in Q.
4. The set of Bloom filters held by each peer is maintained by using the LRU policy.

With this method, the amount of message transmissions could be significantly reduced if the given query has been issued in the past (as will be described later, we can attain a sufficient reduction of data transmission even if the size of B_Q is bounded by 1 KB). In addition, since this scheme checks the existence of Bloom filters for each subconjunction of the given query (Step 2c), it would be effective even when the current query has not been issued in the past.

4.3 Proposed Algorithm

This subsection describes the proposed algorithm. The idea of the proposed algorithm is to reduce the number of indices transferred to the requester in the basic scheme (i.e., Step 4 in BASIC_CACHE) by using Bloom filters as in SIMPLE_FILTER. Concrete procedure is described as follows (the reader should note that different from schemes described in previous subsections, the following scheme takes into account the dynamic change of files, i.e., addition of new indices and removal of existing indices. Such a difference does not affect the simulation result given in the next section, since we will not consider such a dynamic case in the simulation).

Algorithm FILTER_CACHE

- Let $Q = \{w_1, w_2, \ldots, w_x\}$ be a conjunctive query issued by a requester. Steps 1 to 3 are the same with the basic scheme. Thus, we have a set of conjunctions Ψ^* ($\subseteq$ Sub(Q)) after Step 3. Here, note that the value of Size(R) means the number of indices matching conjunction R at the time of the last search for R, which is generally smaller than the number of indices which will pass through Bloom filter B_R concerned with R.
- For each $R \in \Psi^*$, it calculates a set of indices matching R by using a method similar to SIMPLE_FILTER. More concretely, it executes the following operation: 1) for each $w_i \in Q$, obtain a set of conjunctions in Ψ^* containing keyword w_i. 2) send the resulting set of conjunctions to the peer p managing w_i, 3) after receiving it, p eliminates redundant indices in S_i by applying Bloom filters corresponding to the received conjunctions, 4) after taking a union of the resultant set with the set of indices which is newly added to the system after generating Bloom filter used in the above process, and returns the result to the requester.

- After receiving all results, the requester takes an intersection of them, and obtains the search result S for Q.

Note that by taking a union with indices added after the generation of the current Bloom filters, it attains an adaption to the dynamic change of the files (files removed after generating the Bloom filters do not affect the correctness of the scheme, since they do not pass the Bloom filters).

5. Experiment

5.1 Setup

We evaluated the performance of the proposed scheme by simulation. In order to eliminate the effect of the difference of DHT implementations, in the following, we assume that the hash table is a one-dimensional space, and is divided into sub-spaces of an equal size, where each sub-space is assigned to a given set of peers in one-to-one manner. The number of peers is varied from 100 to 1600. The goodness of schemes is evaluated in terms of the number of queries actually submitted to the network and the amount of data transmitted during the query processing. Note that the number of queries reflects the search time (e.g., if we should submit 10 queries to the DHT to process a given conjunctive query, it is 10 times worse than the case in which a single query is enough to acquire the search result), and the amount of transmitted data reflects the traffic over the P2P overlay.

Each peer has 10 files, each of which is associated with at most 10 keywords. Keywords are selected from 1000 candidates according to the Zipf's first law, where each file is associated with 3.5 keywords, on average. We assume that each index or a Bloom filter occupies a unit of memory, where a memory unit corresponds to 1 KB in this experiment, and we will use the number of available memory units in each peer as a parameter. Each query is generated according to a Bernoulli trial with stopping probability $0.2 \leq p \leq 0.5$, and each keyword contained in a query is selected according to the Zipf's law with Zipf parameter 2.0. As was described previously, function Sub is fixed such that: 1) keywords in R are arranged in a lexicographical order, and 2) Sub(R) contains all conjunctions corresponding to the prefixes of the ordered sequence. Finally, for comparison, we used the following two methods as the competitors; i.e., result-caching scheme proposed by Kobatake *et al.* [3] and a query processing scheme proposed by Vahdat [5], which will be referred to as KOB and VAH, respectively (an outline of those schemes will be given in Sect. 6).

5.2 Number of Queries

At first, we evaluate the average number of queries issued to the DHT during the processing of a conjunctive query. In the experiment, we fix the number of peers to 100, the memory size of each peer to 300 units, the total number of keywords to 1000, and the stopping probability of Bernoulli trial to 0.4 (in the following, parameters are determined as above, unless otherwise stated).

Table 1 Average number of query submissions ($|P| = 100$).

Schemes	Average query length			
	3.03	3.51	4.41	6.00
BASIC_CACHE	28.72	141.64	586.44	1332.12
SIMPLE_FILTER	1.25	1.24	1.23	1.22
FILTER_CACHE	14.28	97.39	368.54	1060.52
KOB	1.62	1.53	1.63	1.73
VAH	1.43	1.52	1.63	1.71

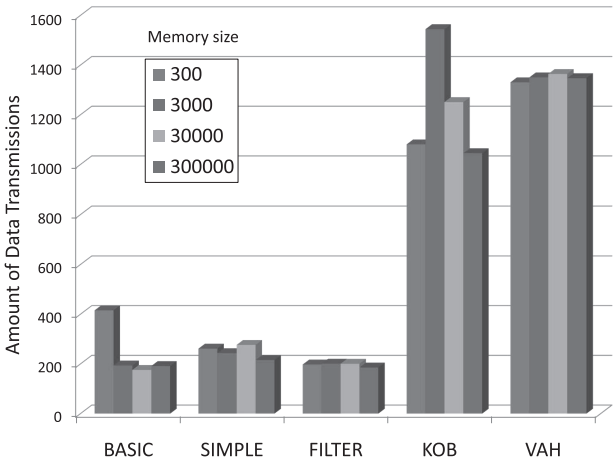


Fig. 3 Impact of memory capacity to the amount of data transmissions.

The result is shown in Table 1. As shown in the table, schemes BASIC_CACHE and FILTER_CACHE transmit much more queries than the other schemes. In fact, 1) the number of queries under KOB is bounded by a small value due to the effect of Bloom filter used to keep the set of conjunctions cached in the DHT (see Sect. 6 for the details), and 2) the number of queries under SIMPLE_FILTER and VAH is smaller than the number of keywords contained in a query, which is because of the effect of "terminating" the evaluation before checking all keywords contained in a conjunction if it becomes apparent that there are no files matching the given query. Such a badness of the proposed schemes in terms of the number of transmitted queries could be improved by combining it with the Kobatabe's scheme although it consumes a large amount of cache space.

5.3 Amount of Data Transmissions

Next, we evaluate the amount of data transmissions per query. In the following, we fix the number of hash functions used in Bloom filters to 10 (recall that this parameter affects the false positiveness of Bloom filters).

5.3.1 Impact of Memory Capacity

We evaluate the impact of the memory capacity to the amount of data transmissions. The result is shown in Fig. 3. Each color corresponds to a given memory size, e.g., "300" indicates that the memory size is 300 units. From the figure, we can find that the amount of data transmissions in

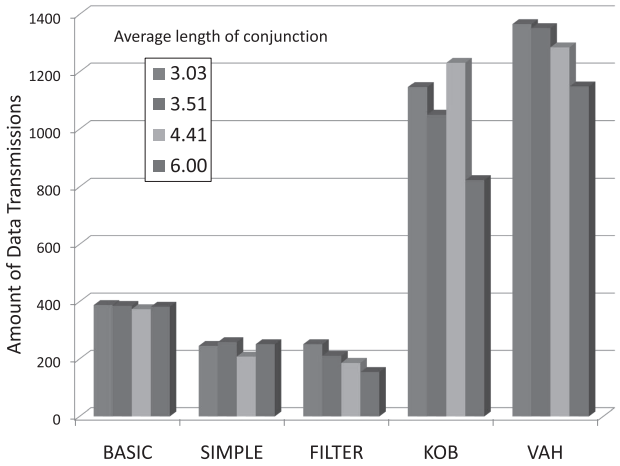


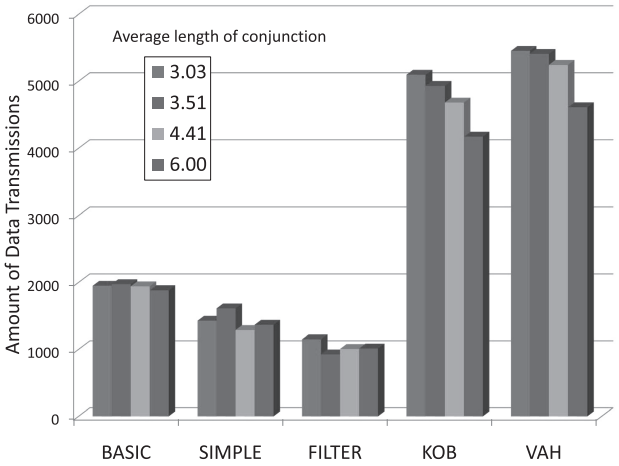
Fig. 4 Amount of data transmissions for $|P| = 100$.

BASIC_CACHE decreases as increasing the memory size of each peer, which is apparently because of the reduction of the frequency of expirations of cached data. However, when the cache capacity reduces to 300 units (=300 KB), FILTER_CACHE beats the other schemes, because of frequent expiration of cached data in previous list-based schemes such as BASIC_CACHE and KOB. Such a frequent expiration causes a re-evaluation of conjunctions by spending a number of (redundant) data transmissions, which enhances the difference between those two schemes.

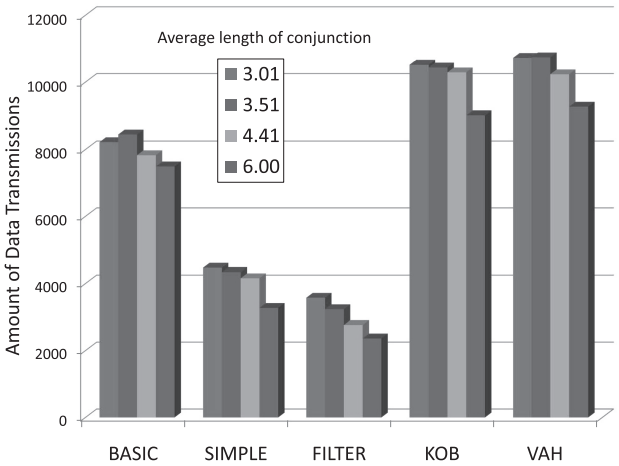
5.3.2 Query Length

In general, the effect of result cache depends on the length of given conjunctive queries. Thus, as the next step, we evaluate the impact of the average query length to the amount of data transmissions by varying the stop probability of Bernoulli trial from 0.2 to 0.5. (The other parameters are the same with previous experiments.) The result is shown in Fig. 4. Each color corresponds to the average query length, e.g., "3.03" indicates that we used a Bernoulli trial such that the average query length is 3.03.

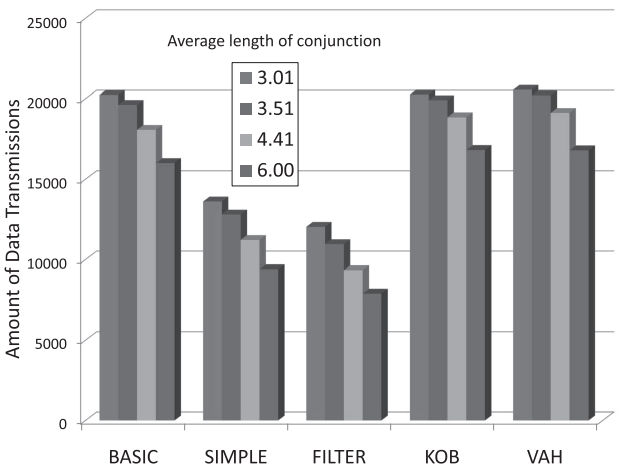
The amount of data transmissions decreases as increasing the query length except for BASIC_CACHE and SIMPLE_FILTER. On average, FILTER_CACHE reduces the amount of data transmissions of KOB by 75% and that of VAH by 80%. Although the improvement from BASIC and SIMPLE is not very large, it still reduces the amount of BASIC by 45% and that of SIMPLE by 15%. A reason of such phenomena is that as increasing the number of keywords contained in a query, the number of files matching the query becomes small. The amount of data transmission for BASIC_CACHE is smaller than that for KOB, which is because of the effect of function Size which tries to select sub-conjunctions in such a way that the amount of resultant data transmissions becomes as small as possible.



(a) $|P| = 400$.



(b) $|P| = 800$.



(c) $|P| = 1600$.

Fig. 5 Scalability of the schemes.

5.3.3 Scalability

We conducted experiments to evaluate the scalability of the schemes. The number of peers is varied from 100 to 1600. The result is summarized in Fig. 5 (the basic set-

ting is the same with Fig. 4). We can see from the figure that the superiority of the proposed schemes does not change even by increasing the number of peers. For example, SIMPLE_FILTER reduces the amount of data transmissions of the basic scheme by 49.7% in the best case, and FILTER_CACHE improves SIMPLE_FILTER by 20.4% in the best case. Another observation we can make from the figure is that the difference between the basic scheme with KOB reduces as increasing the number of peers, which is probably because an increase of the number of peers also increases the number of files matching a given query, which increases the number of conjunctions which cannot be cached with a limited cache memory.

In summary, the proposed scheme is particularly effective to reduce the amount of data transmissions in P2P DHTs with local storage of limited capacity. Such a superiority does not change even if the number of peers in the P2P increases. As for the number of queries, the badness of the proposed schemes can be overcome by combining it with Kobatake's scheme.

6. Related Work

This section overviews previous techniques to realize an efficient processing of conjunctive queries in P2P DHT.

An extension of the hash function to the function from the set of keywords to the hash table was originally proposed by Reynolds and Vahdat [5]. This method is referred to as the reversed DHT method in the literature. Bhattacharjee *et al.* [1] proposed a data structure called view tree to support an efficient processing of conjunctive queries in P2P DHT. View tree realizes a result cache on P2P DHT, and efficiently supports the processing of conjunctive queries by maintaining cached conjunctions in a tree-structured overlay network in a hierarchical manner. More concretely, each vertex in the tree keeps the search result for a conjunctive query similar to our proposed method. The tree is maintained in such a way that the parent vertex corresponds to a conjuction that is a "prefix" of the conjunction corresponding to a child vertex. When a conjunctive query is submitted to a view tree, it first examines if the result for the query is cached in the tree by conduncting a tree traversal in a depth first manner. If it is cached, the requester obtains the search result by simply accessing to the corresponding vertex, and otherwise, the requester inserts a new vertex corresponding to the query to the view tree after obtaining the search result for the query.

Reynolds and Vahdat proposed a way to use Bloom filter to compress the data transmitted between peers while processing conjunctive queries [5]. SIMPLE_FILTER proposed in Sect. 4 is an extension of this scheme. In contrast to our scheme, in their scheme, a recording of file IDs matching a keyword contained in a given query is conducted by each peer holding such indices while conducting a conjunctive search, and in addition, it does not cache the result of conjunctive queries, i.e., Bloom filter is used merely as a tool to transfer a set of indices to the next peer during the processing of cunjunctive queries.

Kobatake et al. proposed another approach to reduce the cost required for the processing of conjunctive queries. In their scheme, a set of indices matching several keywords is cached as in the view tree, and to reduce the time required to find an appropriate conjunction which can be used to process the currently given query, it records the ID of peer holding the result for such conjunction to a Bloom filter and circulates it. Such a circulation among all peers is conducted efficiently by using a tree-structured overlay.

7. Concluding Remarks

In this paper, we proposed a new implementation of result-cache for P2P DHT. The simulation result indicates that the proposed scheme is particularly effective when the memory size is bounded by a small value, and when the number of peers is 100, it reduces the amount of data transmissions of previous schemes by 75% on average. As a future work, we should extend the proposed scheme to a dynamic environment in which peers dynamically join and leave, and we should improve the performance of the scheme by considering the combination of caching scheme and the file search scheme.

References

[1] B. Bhattacharjee, S. Chawathe, V. Gopalakrishnan, P. Keleher, and B. Silaghi, "Efficient peer-to-peer searches using result-caching," Proc. IPTPS 2003, pp.225–236, Feb. 2003.
[2] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," CACM, pp.422–426, July 1970.
[3] K. Kobatake, S. Tagashira, and S. Fujita, "A new caching technique to support conjunctive queries in P2P DHT," IEICE Trans. Inf. & Syst., vol.E91-D, no.4, pp.1023–1031, April 2008.
[4] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," Proc. SIGCOMM, pp.161–172. Aug. 2001.
[5] P. Reynolds and A. Vahdat, "Efficient peer-to-peer keyword searching," Proc. International Middleware Conference, pp.21–40, June 2003.
[6] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), pp.329–350, Nov. 2001.
[7] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," Proc. SIGCOMM, pp.149–160, Aug. 2001.
[8] B. Zhao, J. Kubiatowicz, and A. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Technical Report, UCB/CSD-01-1141, April 2000.

Takahiro Ariyoshi received the B.E. degree in electrical engineering and M.E. degree in information engineering from Hiroshima University in 2008 and 2010, respectively. His research interests include resource management in distributed systems.

Satoshi Fujita received the B.E. degree in electrical engineering, M.E. degree in systems engineering, and Dr.E. degree in information engineering from Hiroshima University in 1985, 1987, and 1990, respectively. He is a Professor at Graduate School of Engineering, Hiroshima University. His research interests include communication algorithms, parallel algorithms, graph algorithms, and parallel computer systems. He is a member of the Information Processing Society of Japan, SIAM Japan, IEEE Computer Society, and SIAM.