Bandwidth-Efficient Continuous Query Processing over DHTs

Yingwu Zhu
Department of CSSE, Seattle University, Seattle, WA
zhuy@seattleu.edu

Abstract

In this paper, we propose novel techniques to reduce bandwidth cost in a continuous keyword query processing system that is based on a distributed hash table. We argue that query indexing and document announcement are of significant importance towards this goal. Our detailed simulations show that our proposed techniques, combined together, effectively and greatly reduce bandwidth cost.

1. Introduction

Continuous queries, different from instantaneous queries by which users retrieve matching documents from existing documents in the system, allows the system to alert "future" matching documents to users. Continuous queries reverse the roles of documents and queries in instantaneous query models. That is, queries are indexed and evaluated against each new document as it is inserted into the system. One example of continuous queries is the *News Alert* feature in Google News. It allows users to register a keyword search query by which the user will be notified via emails of any newly discovered documents matching all the terms in the query.

A distributed continuous query processing system includes four major components: *query indexing*, *document announcement*, *query resolution*, and *document alert*. Query indexing is responsible for indexing continuous queries registered by users. Document announcement announces the appearance of a newly inserted document to nodes which may hold the indexes of relevant continuous queries. Query resolution determines the final set of queries relevant to the new document on these nodes. Document alert notifies users of a newly discovered document matching their interest. Document alert is outside the scope of this paper and it can be implemented by multicast, via emails, or with the help of RSS feeds.

In this paper, we leverage a distributed hash table (DHT) [7, 6, 9] to process continuous queries and focus our study on *simple keyword queries*. Continuous keyword queries are indexed and newly inserted documents are published into the underlying DHT. Each DHT node may serve as a rendezvous between queries and documents in query resolution.

Our primary design goal is bandwidth efficiency. To this end, we investigate various query indexing schemes and propose using multicast techniques to perform document announcement. In particular, we make the following contributions:

- We propose a simple yet powerful indexing scheme MHI to index continuous queries. MHI does not rely on any term statistics and effectively reduces bandwidth cost over random indexing by up to 39.3%.

- We further propose a more efficient query indexing scheme SAP-MHI. SAP-MHI is based on duplicate-sensitive sampling which is a natural fit for DHTs as sampled synopses can be propagated over multiple, redundant DHT paths for both reliability and fast convergence. A moderate sample size of 3,000 terms reduces bandwidth cost over MHI by up to 80.6%.

- We design a multicast-based document announcement mechanism which is performed through embedded trees in the underlying DHT without the need of explicit multicast trees. The document announcement mechanism also allows for optimization of term piggybacking in announcement messages. We argue that query indexing and document announcement are of significant importance in saving bandwidth. We show that, our query indexing schemes and document announcement mechanism, combined together, effectively and greatly cut down bandwidth consumption. To the best of our knowledge, this work is the first to explore the impact of query indexing and document announcement on bandwidth cost in DHT-based continuous query processing systems.

The remainder of the paper is structured as follows. Section 2 provides an overview of related work. Section 3 presents our system model and notations. We discuss query indexing, document announcement, and query resolution

0190-3918/08 $25.00 © 2008 IEEE
DOI 10.1109/ICPP.2008.11

in Section 4, Section 5, and Section 6 respectively. Section 7 evaluates our approach against other approaches. We present an optimization for query indexing as well as experimental results in Section 8. We conclude the paper in Section 9.

2. Related Work

Many query systems on DHTs focus on instantaneous queries: users submit a query and the system returns relevant documents to users. Examples of such systems include simple keyword search [5, 3] and content-based search [8, 10].

There are few distributed continuous query systems in the literature. SmartSeer [2] is the pioneering work to support rich continuous queries over DHTs. It presents the basic architecture of the system and identifies several key issues in query resolution for continuous keyword queries. Our work differs from SmartSeer in that we focus our study on query indexing and document announcement instead of query resolution. We aim to minimize bandwidth cost by aggressively yet effectively pruning the set of queries for query resolution. In fact, SmartSeer and our work complement each other.

Our query indexing scheme through sampling (in Section 8) is inspired by duplicate-sensitive aggregation from reference [4]. Nath et al. presented a duplicate-sensitive approach to aggregate data in sensor networks. We employ the duplicate-sensitive aggregation to sample most popular terms in a DHT and the sample is used to guide query indexing.

3. System Model and Notations

We base our system model on continuous keyword queries. Given a continuous query Q consisting of m unique terms, i.e., $Q = \{t_1, \cdots, t_m\}$, its index is stored into an inverted list corresponding to some term $t_i \in Q$. The inverted list is maintained by a DHT node which is responsible for hash of t_i, i.e., $h(t_i)$. Each index entry of an inverted list includes a query's terms, subscriber's information, alert methods (e.g., via email) and other metadata. Upon a new document D consisting of n unique terms $\{t_1, \cdots, t_n\}$, the system searches all inverted lists corresponding to D's terms and finds the queries whose terms all appear in D. If a query's terms all appear in D, we say that D matches the query. The system will notify the query's subscriber of D via corresponding alert methods.

Notations. DN represents a DHT node that introduces a new document into the system. A DN could be a document insertion node or a node responsible for storing the document. QN denotes a DHT node that maintains inverted lists of continuous queries. Given a new document $D = \{t_1, \cdots, t_n\}$, the DN contacts all QNs that store inverted lists corresponding to D's terms for matching queries. Term ID represents the hash of a term, i.e., $h(t)$. For ease of exposition, we may use *term ID* to represent *term* in the rest of the paper.

4. Query Indexing

Given a continuous keyword query $Q = \{t_1, \cdots, t_m\}$, a straightforward indexing scheme is *random indexing* (RI). By RI, the query is indexed into the DHT under a randomly chosen term $t_i \in Q$; a DHT node responsible for the term ID of t_i (i.e., $h(t_i)$) adds an index entry for Q into the inverted list corresponding to t_i.

Optimal indexing (OI) assumes perfect knowledge of term statistics (i.e., term frequency) in a document corpus, and indexes a query under the most selective term, i.e., the term with the least frequency. Specifically, given a query Q with the most selective term t_i among all its terms, OI stores its index in the inverted list corresponding to t_i on a DHT node responsible for $h(t_i)$. The intuition behind OI is that using the most selective term to index a query allows us to maximize load balancing and minimize bandwidth and processing in query resolution. However, OI is not practical because that, aggregation and distribution of term statistics of documents are nontrivial in peer-to-peer networks where nodes join and leave at will, and where the number of documents can scale to millions and the document collection is subject to changes due to document update, addition and removal.

4.1. Our Approach: MHI

We propose a simple yet powerful indexing scheme called *Minimum Hash Indexing* (MHI). Unlike OI, MHI does not rely on term statistics. Given a query $Q = \{t_1, \cdots, t_m\}$, MHI chooses a term t_i such that its term ID is minimal among all terms, and stores Q's index into the inverted list corresponding to t_i on the node responsible for $h(t_i)$. For ease of exposition, we assume that in the rest of the paper an inverted list corresponding to t_i can be identified by both t_i and its term ID $h(t_i)$ in MHI. Each index entry of an inverted list includes pairs of $(term, termID)$, *sorted in increasing order of term IDs*.

The intuition behind MHI is that only a query that contains a term which appears in the document and whose term ID is minimal among the query's term IDs will be considered in query resolution. Suppose that a query Q has m unique terms, k ($0 \leq k \leq m$) of which appear in a document D. Let P be the probability of that Q is processed in query resolution. Note that P for RI and MHI is: $\frac{k}{m}$ and $\frac{k}{m^2}$, respectively.

Consequently, MHI can filter out many irrelevant queries which otherwise may be considered in query resolution in RI, thereby saving bandwidth and processing in query resolution. As shown in Figure 1, queries Q_1, Q_2, and Q_3 are filtered out by MHI upon announcement of document D, due to the fact that the term with the minimum term ID in

each of the three queries does not appear in D. However, the three queries are very likely (i.e., with probability of 67%) to be considered in query resolution if we use RI.

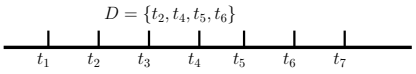


Figure 1. Terms are sorted in increasing order of term IDs. That is, $h(t_i) < h(t_j)$ for $i < j$. There are three queries $Q_1 = \{t_1, t_2, t_4\}$, $Q_2 = \{t_3, t_4, t_5\}$, and $Q_3 = \{t_3, t_5, t_6\}$.

5. Document Announcement

Consider a newly inserted document $D = \{t_1, \cdots, t_n\}$. The DN notifies a set of QNs responsible for $h(t_i)$, where $1 \leq i \leq n$. There are two challenges for document announcement: (1) How does the DN locate all the corresponding QNs of the document? (2) How does the DN intelligently collapse redundant announcement messages to a QN if multiple term IDs fall into the same QN's responsible region [1]?

In real-world applications, a document consists of tens or hundreds of unique terms. For instance, the TREC [1] documents in our experiments had on average 178 unique terms, and the CiteSeer documents used in SmartSeer had on average 2000 unique terms. In a DHT consisting of hundreds or thousands of nodes, the number of QNs could be large, e.g., proportional to the number of document terms. As a result, we base document announcement on multicast techniques. In particular, we exploit embedded trees in the underlying DHT to multicast document announcement messages.

5.1. Using Embedded Trees in DHTs

A DHT like Chord embeds many trees formed by DHT links, more specifically, by neighbor links (i.e., the links to successor nodes and finger nodes). The embedded trees are maintained by periodically performed routing table maintenance messages in the presence of node churn. Consider a DN R and a set of QNs $Z_i, 1 \leq i \leq k$ in Chord. The routing paths from R to each Z_i form a tree rooted at R. For a newly inserted document, R uses this embedded tree to notify each QN Z_i of the document.

We here take a document $D = \{t_1, \cdots, t_n\}$ as an example to discuss how R, as a DN, multicasts the announcement messages to relevant QNs Z_i through the embedded tree. Before multicasting, R groups D's term IDs according to its neighbor. As illustrated in Algorithm 1, term IDs falling into R's responsible region are grouped together (lines 5-7) because R is also a QN for the document in this case. Term IDs falling into a successor node's responsible region are grouped at this successor (lines 9-11); term IDs equal to or immediately following a neighbor's ID are grouped at this neighbor which is either the last successor node or a finger node (lines 12-16). This is based on the observation that *messages with these term IDs as a key are routed through this neighbor in Chord*. The routing paths from R to these term IDs naturally guarantee the location of the corresponding QNs responsible for these term IDs. Moreover, this manner of grouping allows message aggregation and thus minimizes the number of messages. As shown in right component of Figure 5.1, term IDs are grouped according to R's neighbors.

[1] In DHTs like Chord, each node is responsible for a chunk of the DHT identifier space, which we call *responsible region*.

Algorithm 1 R.group_termIDs(Document D)

```
1: vector<ID> id_set[1..k] //stores term IDs with respect to k neighbor nodes
2: vector<ID> local_set //stores term IDs that fall into R's responsible region
3: for each t_i∈D do
4:    ID x = h(t_i) //h() is the consistent hash used in Chord
5:    if x lies in between R's predecessor ID and R's ID then
6:       local_set.push_back(x) //add x into set
7:       continue
8:    end if
9:    find a R's successor node whose ID is equal to or immediately follows x
10:   if such a successor node exists then
11:      j ← the successor node's index entry in R's routing table
12:   else
13:      find a R's neighbor node whose ID is equal to or immediately
         precedes x among all of R's neighbors
14:      j ← the neighbor node's index entry in R's routing table
15:   end if
16:   id_set[j].push_back(x)
17: end for
```

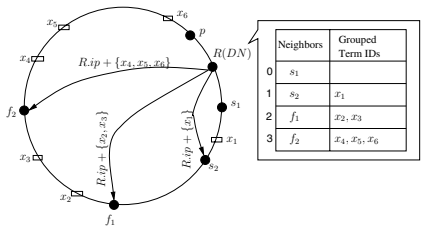


Figure 2. Document announcement from a DN R. s_1 and s_2 are R's successor nodes; f_1 and f_2 are R's finger nodes. p is R's predecessor node. Let $D = \{t_1, \cdots, t_6\}$ and $x_i = h(t_i)$.

After grouping of D's term IDs, R uses $multicast()$ (as shown in Algorithm 2) to send announcement messages each containing a list of term IDs to its neighbor nodes. As shown in Figure 5.1, the message to f_1 encapsulates a list of x_2 and x_3; the message to f_2 encapsulates a list of x_3, x_4, and x_5. Upon receiving the message, Z (here as f_1 or f_2) extracts the term ID list, and regroups the list into $local\_set$ and id_set as shown in Algorithm 3. Z then calls $multicast()$ which delivers $local\_set$ for local processing and $id\_set[i]$ to its i-th neighbor node. This process is repeated recursively along the paths of the embedded tree rooted at R, until all term IDs reach their QNs. We defer discussion of $local\_set$ processing to Section 5.2.

Document announcement, as discussed above, has many advantages: (1) It does not impose overhead of mutli-

Algorithm 2 Y.multicast (vector<ID> $id\_set[1..k]$, vector<ID> $local\_set$)

```
1: if local_set is not empty then
2:    deliver local_set to Y for local processing
3: end if
4: for i = 1 to k do
5:    if id_set[i] is not empty then
6:       Message M ← R.ip + id_set[i] //+ is a concatenation operator
7:       send M to i-th neighbor node Z which calls regroup(M)
8:    end if
9: end for
```

Algorithm 3 Z.regroup (Message M)

```
1: vector<ID> id_set[1..k]
2: vector<ID> local_set
3: vector<ID> recv_set ← extract the term IDs from M
4: for each x ∈ recv_set do
5:    if x lies in between Z's predecessor ID and Z's ID then
6:       local_set.push_back(x)
7:       continue
8:    end if
9:    find a Z's successor node whose ID is equal to or immediately follows x
10:   if such a successor node exists then
11:      j ← the successor node's index entry in Z's routing table
12:   else
13:      find a Z's neighbor node whose ID is equal to or immediately precedes x among all of Z's neighbors
14:      j ← the neighbor node's index entry in Z's routing table
15:   end if
16:   id_set[j].push_back(x)
17: end for
18: Z.multicast(id_set, local_set)
```

cast tree construction and maintenance; (2) It inherits self-organization and fault-tolerance natures of the underlying DHT; (3) Announcement messages are aggregated along the routing paths, thereby minimizing the number of messages; (4) The Proximity Neighbor Selection (PNS) of DHT links naturally enables proximity-aware message multicasting. (5) More importantly, document announcement sends only one message to each QN even if multiple term IDs fall into the QN's responsible region.

5.2. Processing of $local\_set$

As discussed earlier, $local\_set$ is a set of term IDs falling into a QN Z's responsible region. For each term ID $x \in local\_set$, Z searches for the corresponding inverted list. All the inverted lists provide an initial set of queries for query resolution between the DN R and the QN Z, as will be discussed in Section 6.

However, this initial set of queries for query resolution can be pruned by two simple observations. **Observation 1**: For a query Q in this set, if there is a term in Q whose term ID falls into Z's responsible region but does not appear in $local\_set$, then Q can be excluded from this set (Proof by contradiction is omitted). **Observation 2**: If D's maximum term ID is included in each unique document announcement message, then a query Q containing a term whose term ID is larger than the maximum term ID can be excluded from this set.

5.3. Optimization: Piggybacking Successor Term IDs

One argument we make in this paper is that we design MHI and document announcement, combined together, to aggressively prune the set of queries for query resolution, thereby saving bandwidth and processing. In this section, we present an optimization in document announcement by piggybacking to aggressively prune the set of queries for query resolution.

Algorithm 4 Z.prune_invertlist (ID x, vector<ID> $succ\_ids$)

```
1: Z searches for the inverted list corresponding to a term with term ID of x
2: if the inverted list exists then
3:    ID max ← maximum term ID in succ_ids
4:    L ← inverted list
5:    for each Q = {x, x_1, x_2, ···, x_m} in L do
6:       // term IDs are sorted in increasing order, x is the term ID of the indexing term of Q in MHI
7:       for each x_i lies in between (x, max) do
8:          if x_i ∉ succ_ids then
9:             L = L − {Q} //filter out Q
10:            break
11:         end if
12:      end for
13:   end for
14: end if
15: return L
```

For each term ID x in a document announcement message, we piggyback onto the message up to k successor term IDs which immediately follow x in the document. The purpose is to allow each QN to further prune the set of queries for query resolution. The intuition is that hashing terms produces *sparse* term IDs over the DHT identifier space, and thus piggybacking a small number of neighboring term IDs may effectively filter out irrelevant queries.

Algorithm 4 illustrates how a QN Z filters out irrelevant queries in an inverted list corresponding to x with the help of piggybacked successor term IDs (i.e., $succ\_ids$). After optimization, L contains the set of queries for query resolution with respect to this inverted list.

Recall that document announcement messages are aggregated along the routing paths. The term IDs contained in each message are actually *close together* in the Chord ring space. Consequently, the term IDs in each message have many overlaps in their successor term IDs (actually the number of additional term IDs piggybacked is at most k), making piggybacking overhead a lesser issue. k offers a tradeoff between piggybacking overhead and bandwidth cost in query resolution.

6. Query Resolution

Document announcement allows a DN to locate a set of QNs each of which may present a set of queries requiring further examination. Note that the set of queries may have been pruned by the two observations and the optimization of piggybacking, as discussed in Section 5. Query resolution identifies the final set of queries satisfied by a document on each QN. SmartSeer proposes two main techniques for

query resolution: *Term Dialogue* (TD) and *Bloom Filters* (BF). In Term Dialogue, each QN sends a message to the DN asking about the presence of a set of terms in the document, and the DN replies with a bit vector of which each bit specifies the presence of a term. The query resolution may involve multiple rounds of communication between a QN and the DN. In Bloom Filters, the DN sends a bloom filter over all the terms in the document to each of the QNs; each QN filters out queries that contain a term corresponding to a 0 in the bloom filter. By the bloom filter, a QN may prune the set of queries to a smaller one. Then, the QN and DN use Term Dialogue to resolve queries. We adopt the two techniques in our experiments.

7. Evaluation

7.1. Experimental Setup

Table 1. Default parameter values for simulation.

Parameter	Default
Network size	1000 nodes
Document set	TREC-1,2-AP
Mean of query sizes	5
Number of continuous queries	100,000
Number of documents	10,000
Document announcement	w/o piggybacking

Our simulator is built on top of **p2psim 3.0** and uses Chord as the underlying DHT to store queries and documents. Table 1 shows default parameter values for simulations. We index $100,000$ continuous queries into the system. Then, each document in the document set is inserted into the system, which triggers document announcement and query resolution. The results are collected and averaged over the whole document set. Simulations are based on a document set *TREC-1,2-AP* [1]. The document set used in our experiments consists of $10,000$ documents. We use SMART to extract unique terms from each document. The extracted terms are stemmed and stop words are removed. This results in $46,654$ unique terms for the document set. Each document on average has 178 unique terms.

There are few continuous query workloads available to us, so we base our simulations on a workload of synthetically generated queries. Before generating queries, we calculate the term frequency distribution over the document set. We then generate three categories of queries: *Uniform*, *Skew*, and *InverSkew*. The number of terms in a query follows a normal distribution with mean v and standard deviation $0.3v$. In *Uniform*, each query chooses its terms uniformly from the document set, irrespective of term frequency. In *Skew*, each query chooses its terms according to the term frequency distribution, i.e., those terms that appear frequently in documents also appear frequently in queries. In *InverSkew*, each query chooses its terms by inverting the skew, i.e., those terms that appear frequently in documents appear infrequently in queries.

Our simulator has three query indexing components, namely RI, OI, and MHI. The calculated term frequency over the document set is used by OI to index continuous queries. Query resolution uses two techniques, namely *Term Dialogue* and *Bloom Filters*. When using Bloom Filters in query resolution, we use a bloom filter of $1K$ bits and five independent hash functions.

We use three metrics for evaluation: (1) *Number of queries per document considered for query resolution* which defines how well a query indexing scheme (and document announcement) filters out irrelevant queries before query resolution. (2) *Bandwidth cost per document* in query resolution, which also indicates a query indexing scheme's (and document announcement's) ability of filtering out irrelevant queries before query resolution. (3) *Load distribution* specifies query index distribution among nodes. Specifically, we use 1st, 50th, and 99th-percentiles of number of query indexes on nodes.

7.2. Results

In this section, we present experimental results to answer two main questions: (1) How well is MHI able to discard irrelevant queries before query resolution and thus save bandwidth? (2) How will the three query indexing schemes perform under different query types and query resolution techniques?

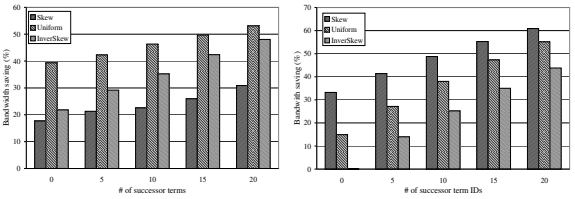
Table 2. Results for Skew Queries.

Indexing	Load dist.	# of queries/doc	Bandwidth (Bytes)	
			TD	BF
RI	(4, 52, 706)	11476	282285	36853
MHI	(0, 18, 1027)	8560	188530	30338
OI	(8, 67, 487)	649	12518	20464

Basic comparisons.
Table 2 shows results for Skew queries. Several observations can be made form this table: (1) OI handles load balancing best while MHI incurs most load imbalance. This means some popular terms happen to have minimum term IDs in continuous queries, thus some nodes are responsible for long inverted lists corresponding to these popular terms. (2) MHI filters out 25.4% more irrelevant queries than RI, thereby reducing bandwidth cost in query resolution by 33.2% and 17.7% for Term Dialogue and Bloom Filters respectively. This shows that MHI, while simple, can effectively save bandwidth for continuous query processing. (3) OI filters out 94.3% more irrelevant queries than RI, thereby saving bandwidth over RI by 95.6% and 44.5% for Term Dialogue and Bloom Filters respectively. This confirms our intuition that indexing queries under the most selective term is most effective in both bandwidth saving and load balancing.

Due to space constraints, the results for Uniform and InverSkew queries are omitted here. The results for InverSkew are similar to Skew except that MHI saves little bandwidth over RI for Term Dialogue. This is mainly because most terms in InverSkew queries are rare across the documents and there are good indexing term candidates. Thus, RI performs very well and renders MHI not much room to improve performance. For Uniform queries, MHI reduces bandwidth over RI by 14.9% and 39.3% for Term Dialogue and Bloom Filters respectively.

Impact of Piggybacking Successor Term IDs.
Figure 3 shows bandwidth savings by MHI over RI with respect to the number of piggybacked successor term IDs in document announcement messages. Note that piggybacking successor term IDs effectively and significantly reduces bandwidth cost. This is largely because that hashing terms produces sparse term IDs over the DHT identifier space. As a result, piggybacking a small number of neighboring term IDs can effectively filter out irrelevant queries before query resolution. The piggybacking overhead is small since a term ID only consumes several bytes, e.g., 8 bytes in our simulations.



(a) (b)

Figure 3. Bandwidth saving by MHI over RI with respect to the number of piggybacked successor term IDs. (a) Bloom Filters. (b) Term Dialogue.

8. Query Indexing Optimization: SAP-MHI

Previous experimental results suggest that it be wise to index queries under the most selective terms. MHI neglects term statistics and thus may incidentally index queries under popular terms, despite great improvement over RI. As a result, we may improve MHI's performance by avoiding indexing queries under popular terms. To this end, we optimize MHI by sampling K most popular terms in existing documents. K offers a tradeoff between bandwidth overhead and sample size. The basic idea is that we first produce *synopses* by sampling K most popular terms of existing documents, and then each node uses the synopses to guide query indexing. Sampling only affects query indexing (and pruning of initial query set upon document announcement messages if successor term IDs are piggybacked in the announcement messages) as will be discussed later.

8.1. Duplicate-Sensitive Sampling

One challenge for the synopsis sampling is to support *duplicate-sensitive aggregates* as a synopsis may be gossiped over multiple DHT overlay paths and the term document frequency is thus overestimated in aggregates. Duplicate-sensitive sampling needs to repress duplicated synopses in aggregates. We borrow the idea of duplicate-sensitive aggregation from Nath et. al [4].

The goal of duplicate-sensitive sampling is to produce a synopsis that contains up to K most popular terms without being inflated. The sampling algorithm is based on the *coin tossing experiment CT(y)*: toss a fair coin until either the first head occurs or y coin tosses end up with no head, and return the number of tosses. In this paper, we set y to be larger than $\log N$ where N is the number of DHT nodes.

Initially, each node produces a synopsis from their locally stored documents as follows: for each unique term t in a document, the node produces a value $v = CT(y)$ for t. Then, the node aggregates terms over the local documents. Consider a term t with values $v_1, \cdots, v_m$ in documents $D_1, \cdots, D_m$ respectively, the node produces a pair $(t, max(v_1, \cdots, v_m))$ for the term. The synopsis contains up to K pairs of (t, v) with highest values. Then, the node gossip the synopsis to its neighbor nodes.

Upon receiving a synopsis s', each node aggregates its own synopsis s with s'. For each unique term t in $s \cup s'$, discard all but the pair (t, v) with maximum value. Then, the node uses the K pairs with highest values as the current synopsis. Periodical gossip messages are used to refresh each node's synopsis. The intuition behind the duplicate-sensitive sampling is that, if a term appears in more documents then its value produced by $CT(y)$ will be larger than the values of rare terms.

8.2. Adapting MHI to SAP-MHI

MHI needs to adapt its indexing algorithm to exploit the sampled synopsis which we call *SAP-MHI*. Algorithm 5 illustrates how a node Z indexes a query Q into the system using SAP-MHI. The basic idea behind SAP-MHI is to avoid indexing a query under a popular term. Note that synopses on each node are not tightly synchronized. This does not affect the correction of system.

Algorithm 5 Z.SAP_MHI(Query Q)

```
1: sort terms in Q = {t_1, ..., t_m} such that their term IDs are in increasing order, i.e., h(t_i) < h(t_j) for 1≤i < j≤m
2: S ← extract terms from Z.synopsis
3: idx_term = t_1  //the default indexing term in MHI
4: for i = 1 to m do
5:    if t_i ∉ S then
6:       idx_term = t_i
7:       break
8:    end if
9: end for
10: index Q into a node which is responsible for h(idx_term).
```

If we piggyback successor term IDs in document an-

nounce messages (in Section 5), we need to adapt algorithm 4 in pruning an inverted list. Because of SAP-MHI, there are two different presentations of queries in an inverted list: *regular* and *special*. A regular query is one whose indexing term is the term with the minimum term ID, while a special query is one whose indexing term is not with the minimum term ID. For ease of exposition, we represent the special query as $Q = \{x_1, \cdots, x_i, x, x_j, \cdots, x_m\}$ where term IDs are sorted in increasing order and x is the indexing term ID (each term ID is associated with the term). The regular query is denoted as $Q = \{x, x_1, \cdots, x_m\}$ where term IDs are sorted in increasing order and x is the indexing term ID. We handle regular queries as usual but treat special queries differently. For a special query, SAP-MHI leaves the resolution of the popular term IDs $x_1, \cdots, x_i$ to query resolution, as these terms are likely to appear in a document. Algorithm 6 illustrates how to prune an inverted list given special and regular queries due to SAP-MHI.

Algorithm 6 Z.prune_invertlist (ID x, vector<ID> $succ\_ids$)

```
1: Z search for the inverted list corresponding to a term with term ID of x
2: ID max ← maximum term id in succ_ids
3: if the inverted list exists then
4:   L ← the inverted list
5:   for each Q∈L do
6:     if Q = {x, ···, x_m} then
7:       //regular queries
8:       for each x_i lies in between (x, max) do
9:         if x_i ∉ succ_ids then
10:          L = L − {Q} //discard Q
11:          break;
12:        end if
13:      end for
14:    end if
15:    if Q = {x_1, ···, x_i, x, x_j, ···, x_m} then
16:      //special queries
17:      for each tid∈{x_j, ···, x_m} and tid < max do
18:        if tid ∉ succ_ids then
19:          L = L − {Q}
20:          break;
21:        end if
22:      end for
23:    end if
24:  end for
25: end if
26: return L
```

8.3. Re-indexing Queries

Each node keeps refreshing their synopses through gossips, though the interval could be large. As documents are inserted or removed, old terms may vanish and new terms may appear in the current synopsis. Each node can independently re-index the queries in their inverted lists to reflect the flux of popular terms. Currently, we do not incorporate this idea into our design.

8.4. Experimental Results

In this section, we present experimental results to answer the following questions: (1) How much does SAP-MHI improve performance over MHI in terms of load balance and bandwidth saving? (2) Does a small synopsis work well? (3) What gains are achieved by the combination of SAP-MHI and document announcement which piggybacks successor term IDs into messages?

In simulations, we first insert a set of $5,000$ documents into the system. Then, we sample K most popular terms. For each query, we randomly choose a node to index the query using SAP-MHI, MHI or RI. After indexing $100,000$ queries, we insert the second set of $5,000$ documents into the system. The results are collected from the second set of documents.

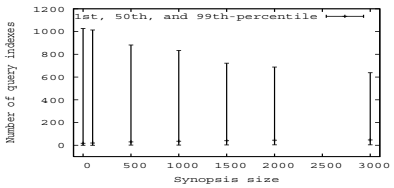
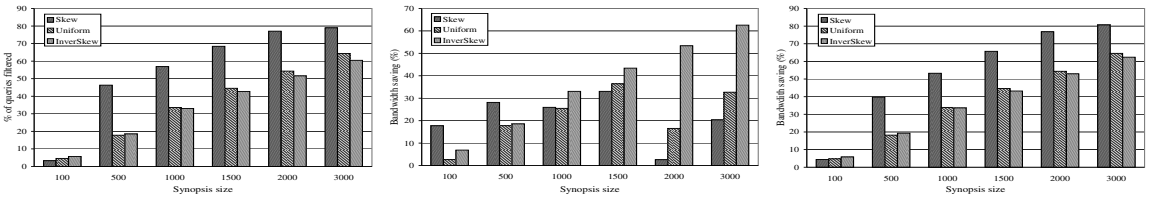


Figure 4. 1st, 50th, and 99th-percentiles of number of queries among nodes for Skew queries.

Figure 4 plots the load distribution with respect to the synopsis size for Skew queries. We omit the figures for Uniform and InverSkew queries due to space constraints. We make the following observations: (1) As the synopsis size increases, SAP-MHI improves load balance greatly on Skew queries. This is because SAP-MHI avoids indexing queries under the most popular terms. (2) SAP-MHI shows little improvement on Uniform and InverSkew queries with increasing synopsis size. For InverSkew queries that contain rare terms, SAP-MHI exploiting popular terms does not affect index distribution. For Uniform queries that contain terms uniformly from a large set of terms $(46,654)$, even a synopsis size of 3000 (6.4% of terms) does not affect index distribution much.

Figure 5 shows percentage of reduction by SAP-MHI over MHI in the number of queries presented to query resolution and bandwidth cost, with respect to the synopsis size. Several observations can be made: (1) With a small synopsis size, SAP-MHI significantly prunes queries, resulting in a much less number of queries to resolve than MHI, i.e., up to 78.9% reduction in number of queries over MHI. (2) SAP-MHI greatly reduces bandwidth cost over MHI due to a less number of queries processed in query resolution, i.e., up to 80.6% bandwidth saving over MHI. (3) SAP-MHI shows an anomaly on Skew and Uniform queries when Bloom Filters are used in query resolution. For example, a synopsis of $2,000$ *surprisingly* shows less bandwidth saving than a synopsis of $1,500$. This is counter-intuitive because a lager synopsis size results in less number of queries to resolve, as shown in Figure 5(a). However, note that with a larger synopsis size, SAP-MHI better distributes query indexes. Consequently, a synopsis of $2,000$ results in more QNs than a synopsis of $1,500$. In other words, a DN communicates with more QNs for query resolution with a syn-



(a) (b) (c)

Figure 5. % of reduction by SAP-MHI over MHI. (a) # of queries per document for query resolution . (b) Bandwidth cost by Bloom Filters. (c) Bandwidth cost by Term Dialogue.

opsis of $2,000$, thereby incurring more bloom filters used in query resolution. Bloom filters are only used to prune the set of queries on each QN. The overhead in bloom filters exceeds the gain in pruning the set of queries. Note that a synopsis of $3,000$ further reduces the number of queries for resolution on each QN, thus compensating the overhead in bloom filters to some extent. While counter-intuitive, this anomaly is encouraging. This is because a small synopsis (i.e., $1,500$, 3.2% of terms) performs very well for Skew and Uniform queries when using bloom filters in query resolution.

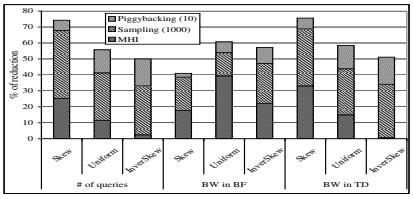


Figure 6. Breakdown of % of reduction by SAP-MHI w/ piggybacking up to 10 successor term IDs over RI. The synopsis size is 1000.

Figure 6 shows percentage of reduction by SAP-MHI with piggybacking over RI, represented by the whole columns where each component indicates the contribution by MHI, sampling and piggybacking, respectively. There are three main observations: (1) By sampling a small synopsis (i.e., $1,000$), SAP-MHI significantly reduces the number of queries for query resolution and bandwidth cost for all three query categories. (2) MHI shows a little bandwidth reduction for InverSkew queries when using Term Dialogue. This is because query terms are rare terms in general, rendering little room of improvement for MHI. (3) Piggybacking a small number (i.e., 10) of successor term IDs in document announcement messages are more effective in Uniform and InverSkew queries. In summary, all the techniques, combined together, significantly reduce bandwidth cost in processing continuous queries.

9. Conclusions

In this paper we present novel query indexing schemes and document announcement to minimize bandwidth cost in a continuous keyword query processing system. We discuss several query indexing schemes and provide a detailed comparison of these schemes on bandwidth cost and load balancing. We point out that query indexing and document announcement are of significant importance in saving bandwidth and processing. We show that our proposed techniques effectively and greatly reduce bandwidth cost in continuous query processing.

References

[1] Text retrieval conference (trec). http://trec.nist.org.

[2] J. Kannan, B. Yang, S. Shenker, P. Sharma, S. Banerjee, S. Basu, and S.-J. Lee. Smartseer: Using a DHT to process continuous queries over peer-to-peer networks. In *Proceedings of IEEE INFOCOM*, 2006.

[3] J. Li, B. T. Loo, J. Hellerstein, F. Kaashoek, D. R. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems(IPTPS)*, pages 207–215, Berkeley, CA, Feb. 2003.

[4] S. Nath, P. B. Gibbons, S. Seshan, and Z. R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, New York, NY, USA, 2004.

[5] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference (Middleware)*, pages 21–40, Rio de Janeiro, Brazil, June 2003.

[6] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed System Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, Nov. 2001.

[7] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM*, pages 149–160, San Diego, CA, Aug. 2001.

[8] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proceedings of ACM SIGCOMM*, pages 175–186, Karlsruhe, Germany, Aug. 2003.

[9] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerance wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, University of California, Berkeley, Apr. 2001.

[10] Y. Zhu and Y. Hu. Efficient semantic search on DHT overlays. *Journal of Parallel and Distributed Computing*, 67(5):604–616, May 2007.