The VLDB Journal (2011) 20:793–817
DOI 10.1007/s00778-011-0224-z

REGULAR PAPER

STAIRS: Towards efficient full-text filtering and dissemination in DHT environments

Weixiong Rao · Lei Chen · Ada Wai-Chee Fu

Received: 23 April 2010 / Revised: 8 December 2010 / Accepted: 10 February 2011 / Published online: 9 March 2011
© Springer-Verlag 2011

Abstract Nowadays "live" content, such as weblog, wikipedia, and news, is ubiquitous in the Internet. Providing users with relevant content in a timely manner becomes a challenging problem. Differing from Web search technologies and RSS feeds/reader applications, this paper envisions a personalized full-text content filtering and dissemination system in a highly distributed environment such as a Distributed Hash Table (DHT) based Peer-to-Peer (P2P) Network. Users subscribe to their interested content by specifying input keywords and thresholds as filters. Then, content is disseminated to those users having interest in it. In the literature, full-text document publishing in DHTs has suffered for a long time from the high cost of forwarding a document to home nodes of all distinct terms. It is aggravated by the fact that a document contains a large number of distinct terms (typically tens or thousands of terms per document). In this paper, we propose a set of novel techniques to overcome such a high forwarding cost by carefully selecting a very small number of meaningful terms (or key features) among candidate terms inside each document. Next, to reduce the average hop count per forwarding, we further prune irrelevant documents during the forwarding path. Experiments based on two real query logs and two real data sets demonstrate the effectiveness of our solution.

W. Rao · L. Chen (✉)
Department of Computer Science and Engineering,
The Hong Kong University of Science and Technology,
Hong Kong, China
e-mail: leichen@cse.ust.hk

W. Rao
e-mail: wxrao@cse.ust.hk

A. W.-C. Fu
Department of Computer Science and Engineering,
The Chinese University of Hong Kong, Hong Kong, China
e-mail: adafu@cse.cuhk.edu.hk

Keywords Content filtering · Content dissemination · DHT

1 Introduction

The amount of *live* content in the Internet such as weblog, Wikipedia, and news is growing at an amazing speed. For example, in July 2006, there were over 1.6 million blog postings every day; the number of blogs worldwide was reported to be 50 million and has doubled every 6 months [30]; in Wikipedia, which has 2,163,836 articles in the English version, the increase in average additions per day during the period from 2002-01-01 to 2007-01-01 was from 54 to 1,822 [2]. These staggering numbers suggest a significant shift in the nature of Web content from static pages to continuously created and updated documents.

With such "live" content, providing users with their interested content in a timely manner becomes a very helpful but challenging task. Web search technologies retrieve Web documents for input queries; however, they are ill-suited to notify users of rapidly changing content [16]. Besides, alert services (like Google Alerts [3] and Microsoft Live Alerts [1]) allow end users to input keywords and receive relevant Web content. However, it is believed that alert services are still based on batch processing through search engines [16]. Thus, they cannot offer timely dissemination services.

Specially, we use the recently popular RSS technology as a motivation example. With help of RSS URLs, end users subscribe to their favorite content. RSS readers *periodically* download RSS feeds containing a summary of new articles that are recently posted to the websites of URLs. Then, subscribers are notified of these new articles. However, the RSS technology suffers from the following two issues.

- The scalability issue due to the high bandwidth consumption: RSS readers adopt a periodic polling mechanism to detect new articles. Given a Web server subscribed by a large number of RSS readers, the network bandwidth consumption by the periodical polling mechanism is illustrated by the following examples [34]. In the *New York Times*, the total network bandwidth is around 3.5 GB/day, for RSS polling the front page every 30 min. For websites wishing to provide RSS readers with deeper content and faster polling frequency, the problem becomes worse. For example, *Boing Boing*, having 11,500 subscribers who subscribe to the provided RSS and Atom with complete HTML stories (40 KB for each RSS request), has to accommodate 22 GB/day of RSS traffic alone. Assuming that the *BBC News* Web site is truly "updated every minute", the RSS polling (given 18,000 RSS subscribers to its various feeds on Bloglines [34]) demands a virtually impractical bandwidth of 989 GB/day.
- The second issue of the RSS technology is that it provides limited filtering semantics. By entering URL addresses of RSS feeds, RSS readers download articles posted to the Web servers associated with those URLs. Now, the problem is, *all* posted articles, though some of them are totally irrelevant to subscribers, are downloaded by RSS readers which then alter subscribers. Following the example above, due to a large number of articles posted to the front page of *New York Times*, if without any filtering mechanism, subscribers could be overwhelmed and annoyed by the amount of useful and useless content. Therefore, a content filtering mechanism is quite necessary to filter out useless content.

In view of the shortcomings of previous approaches, such as no timely dissemination, no content-based filtering mechanism, and scalability issue, we aim to develop a personalized full-text filtering and dissemination system in a large distributed environment, such as a Peer-to-Peer (P2P) network. In such a system, end users subscribe to their interested content by entering input *keywords* as filtering conditions (i.e., subscription *filters*). The system then disseminates relevant content that matches filters to subscribers who want such content. Building this system in a P2P network aims to save the network bandwidth consumption with help of a large number of decentralized peer nodes. Thus, this system overcomes the limited scalability issue as shown in the RSS architecture. Note that such a system is not designed to totally replace RSS technique or Google-like search engines, but instead serve as a complementary service (just like Google Alert Service), and it can even benefit from the RSS technology, for example, customizing the widely familiar RSS reader interface to enter input keywords [24].

For the content-based filtering mechanism, we employ the standard Vector Space Model (VSM) [5] model by using a score function such as *term frequency* * *inverse document frequency* (*tf***idf*) scheme to measure the relevance score between a publication document and a subscription filter. Only those documents having relevance scores higher than a threshold are disseminated to users. In this paper, both input keywords and thresholds are used as filtering conditions, based on two observations: (i) term input has become the de facto standard for end users to retrieve their interested documents; (ii) the whole content space is extremely huge, whereas input queries are usually composed of several terms (on average smaller than 3 terms as shown by real data sets in our experiments). Therefore, if only query terms are used as filtering conditions, without setting a threshold, matching results will be very large. Moreover, end users are usually interested in documents that are of interest to them. Thus, it is quite important for end users to set thresholds to filter out useless documents and only receive relevant documents.

1.1 Data model

In this paper, we consider two data models: a simple model and an extended model.

Simple model: We assume that each document is represented by a set d consisting of $|d|$ pairs of $\langle t_i, s(t_i, d)\rangle$, where t_i is a term, and $s(t_i, d)$ represents the score or importance of t_i in the document. $s(t_i, d)$ can be pre-computed, e.g., using the *tf***idf* scheme. For simplicity, we use the notation of d to indicate both a document and a set of the aforementioned pairs associated with this document.

Each subscription filter is represented by a predefined threshold and a set of $|f|$ terms $\{t_1, .., t_{|f|}\}$. Similar to d, the notation f is used to indicate both a filter and a set of query terms associated with this filter. The predefined threshold, denoted by $\mathrm{T}(f)$, can be specified with either a default value T or a personalized value. Document d is disseminated to the subscriber specifying filter f, provided that

$$\mathrm{S}(f, d) = \sum_{i=1}^{|f|} s(t_i, d) \geq \mathrm{T}(f) \quad (1)$$

In the above equation, $\mathrm{S}(f, d)$ computes the relevance score between f and d. If a subscriber does not specify a threshold, the system assumes that the associated filter uses a *default* threshold T (that works well for general users). The personalized threshold indicates the strength of the desired relevance between input keywords and a document to be received. A higher value of $\mathrm{T}(f)$ indicates the expectation of more relevant documents, and vice versa. $\mathrm{S}(f, d) = 0.0$ means that there is no term appearing in both f and d. By default, all algorithms and statements in this paper will refer to this simple model.

Extended model: In this model, each term $t_i \in f$ is associated with a weight, $w(t_i, f)$, to indicate a subscriber's preference of t_i. Then, Eq. 1 is extended as follows.

$$S(f, d) = \sum_{i=1}^{|f|} [nw(t_i, f) \cdot ns(t_i, d)] \geq T(f) \quad (2)$$

In the above equation, $nw(t_i, f) = \frac{w(t_i, f)}{\sqrt{\sum_{i'=1}^{|f|}[w^2(t_{i'}, f)]}}$ and $ns(t_i, d) = \frac{s(t_i, d)}{\sqrt{\sum_{i'=1}^{|d|}[s^2(t_{i'}, d)]}}$ indicate the normalized weight of t_i in f and normalized score of t_i in d, respectively. Thus, $S(f, d)$ above is a normalized value within [0.0, 1.0]. Based on this model, $S(f, d) = 1.0$ indicates that document d is exactly the same as filter f, including the same terms inside both d and f, and term scores $s(t_i, d)$ equal to corresponding weights $w(t_i, f)$. In addition, $S(f, d) = 0.0$ indicates the same situation as the simple model. For simplicity, we assume that $w(t_i, f) > 0$ is satisfied for each term t_i in filter f.

1.2 Problem statement and challenges

Problem statement: Given two data models above, we want to design a full-text filtering and dissemination scheme in a distributed hash table (DHT) based P2P network, satisfying the following requirements:

User requirements include (i) no or low false dismissals (i.e., subscribers should correctly receive expected documents satisfying either of two data models); and (ii) a timely manner (i.e., subscribers expect to timely receive fresh documents).

Efficiency requirement indicates a low document publishing cost. The publishing cost is measured by the overall number of hops to forward every document to all satisfied subscribers.

Challenges: The challenge is designing a desirable scheme that satisfies all above requirements.

If we relax either of the user requirements, we achieve a useless, though efficient, scheme. For example, if a *high false dismissal ratio* is allowed, we can heuristically forward document d to some randomly chosen nodes. This heuristic scheme consumes a low forwarding cost, but subscribers may not receive useful documents. Or if the *timely requirement* is not emphasized, we can adopt the full-text search scheme [48], where documents are pre-stored in a DHT. After that, searching documents only involves the communication between the query initiator and home nodes of the query terms appearing in filter f. However, this scheme only finds outdated documents, instead of those documents as fresh as possible.

Now, the challenges of designing an *efficient* scheme meanwhile satisfying user requirements are illustrated as follows. First, naively broadcasting document d to all nodes in a DHT incurs the lowest efficiency. Next, an improvement over such a naive solution is to forward document d only to the home node of every distinct term inside d [48,17,22]. However, this approach still suffers from a high publishing cost. For example, it has been shown that the bandwidth consumption of full-text document publishing in the DHT is six times of the super-peer system [48]. The high publishing cost depends on two factors: (i) the cost of forwarding document d to one destination home node (called the *unit-publishing cost*), typically equal to $O(\log N)$ hops, where N is the total number of nodes in a DHT; (ii) a large number of distinct terms per document (called the *publishing amount*). For example, in a data set of our experiments, there are thousands of terms per document. In particular, [48,17,22] store document d (more precisely, the pointer of a raw document d) to home nodes of all distinct terms inside this document. As a result, the total publishing cost, related to the multiplication of two aforementioned factors, is very high.

In addition, the approach, by which every node in a DHT stores the details of all filters, incurs a non-trivial maintenance cost. For example, if a subscriber adjusts his/her personalized threshold $T(f)$, all N nodes in the DHT have to maintain a consistent value of $T(f)$. Otherwise, the subscriber receives either more or less documents than expected. Moreover, because the details of all filters are available in each node, this approach easily leaks private subscription filters defined by subscribers [35,36,27].

1.3 Our approach

First, we propose to register filter f to the home node of *every* term in f, called *full registration*. It avoids the high maintenance cost of registering f to all N nodes.

Consider a node which wants to publish document d. When filter f is not registered to this node, the exact value of $S(f, d)$ is unknown to this node. Without such an exact value, we instead propose a set of techniques to approximate $S(f, d)$ via various filter information. After that, via the approximated value of $S(f, d)$, we select a small number of meaningful terms. Thus, document d is forwarded to a few home nodes, called *partial forwarding*. Due to partial forwarding, we reduce the publishing amount. We call the threshold-based full-text filtering and dissemination system that is enabled by full registration and partial-forwarding STAIRS.

Next, to reduce the unit-publishing cost, STAIRS discards irrelevant documents during their forwarding paths toward destination home nodes. When both the unit-publishing cost and the publishing amount are reduced, the overall publishing cost is significantly low.

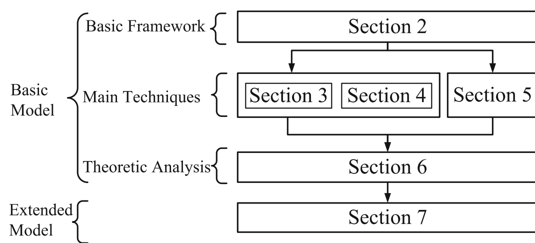


Fig. 1 Overall logic flow chart

In STAIRS, though filter f is fully registered to $|f|$ home nodes, we design a *stateless* mechanism to avoid duplicate document notification. By such a *stateless* mechanism, there is no need to remember the history of those already-notified documents. In summary, the contributions of this paper include:

- Full-text document publishing for a long time has suffered from a high publishing cost [48,17,22]. In this paper, we propose a set of novel techniques to overcome the problem of the high publishing cost by carefully selecting a small number of meaningful terms (or key features) among tens or thousands of candidate terms.
- We propose techniques to reduce the average routing hop count per forwarding, which is smaller than $O(\log N)$, by discarding irrelevant documents during the forwarding path toward destination nodes.
- With real query logs and document corpus, our experimental results verify our analytical findings.

The rest of this paper is organized as follows: Sect. 2 shows the basic framework. As the main techniques, Sect. 3 (assuming every filter adopts a *default* threshold) and Sect. 4 (allowing every filter uses a *personalized* threshold) design algorithms to reduce the publishing amount, and Sect. 5 proposes the scheme to reduce the unit-publishing cost. Section 6 analyzes proposed algorithms. Next, Sect. 7 focuses on the extended data model (The logic flow chart of Sects. 2–7 is shown in Fig. 1). After that, Sect. 8 investigates related works, and Sect. 9 evaluates our algorithms. Finally, Sect. 10 concludes this paper.

2 Basic framework

In this section, we first review DHT-based P2P Networks and then give the framework of STAIRS.

2.1 Overview of DHT-based P2P networks

A number of structured P2P routing protocols have been recently proposed, including Chord [37], Pastry [31], Tapestry [49], etc. These P2P systems provide the functionality of a scalable distributed hash table (DHT), by reliably mapping a given object key (e.g., a term with the string type) to a unique live node in the network. For simplicity, in this paper, we call the node, which is responsible for the object key, the *home node* of this key. These DHT systems have the desirable properties of high scalability, fault tolerance, and efficient routing of queries. Typically, each node is assigned with $O(\log N)$ links, and the average number of routing hops (in short *hop count*) is guaranteed with $O(\log N)$.

2.2 Filter registration

Each end user subscribes to his/her interested documents by a filter f consisting of $|f|$ terms $\{t_1, ..., t_{|f|}\}$ and a threshold $\mathrm{T}(f)$. Here, we define the *size of filter* f by the number of terms in f, i.e., $|f|$.

When a subscriber sends a subscription request containing filter f to STAIRS, f is registered in the home node of *every* term $t_i \in f$. Thus, f is registered to $|f|$ home nodes. We call this "*full registration*". In Fig. 2, filter f_2 is registered to three home nodes respectively for B, D, and E; f_3 is to three home nodes respectively for C, E, and F.

2.3 Document publication

First, we show the basic idea of publishing documents. In the simple model of Sect. 1.2, only those terms, *commonly* appearing in both filter f and document d, contribute to $\mathrm{S}(f, d)$. That is, if a term $t_i \in f$ does not appear in d, $s(t_i, d)$ makes no contribution to $\mathrm{S}(f, d)$. Therefore, after filter f is registered to $|f|$ home nodes, the home node of such a common term can act as the *rendezvous node* for the matching between registered filters and forwarded documents. In particular, to publish a document d containing $|d|$ pairs of $\langle t_i, s(t_i, d)\rangle$ $(1 \le i \le |d|)$, two steps are required:

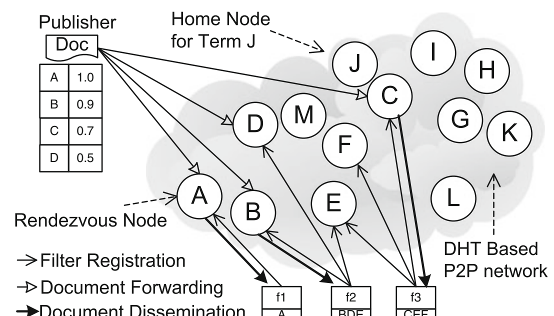


Fig. 2 Basic STAIRS Framework

– *Forwarding*: d is forwarded to home nodes of some selected (meaningful) terms in d;
– *Notification*: if the condition $S(f, d) \geq T(f)$ is satisfied, the subscriber specifying f is notified.

In the first step, the forwarding cost mainly depends on two factors: the number of selected terms and the cost to forward a document to the home node of a selected term. A simple approach is to select all $|d|$ distinct terms from document d and forward d to $|d|$ home nodes. However, the cost of this approach is very expensive, because (i) the number of distinct terms in d, i.e., $|d|$, is large; and (ii) the average hop count of forwarding d to the home node of a selected term is $O(\log N)$. In the second step, filter f satisfying the condition $S(f, d) \geq T(f)$ is called a "*qualified filter*". Given a document d, there could exist multiple qualified filters. All subscribers specifying these qualified filters are notified. It is obvious that the notification cost mainly depends on the number of qualified filters. In this case, the application-level multicast helps save the network bandwidth [33].

2.4 Notification without duplicates

Though this paper focuses on the forwarding phase, an important issue in the notification phase is to avoid duplicate notifications. This subsection presents the technique of notification without duplicates. When document d reaches the home node of a selected term, for every filter f locally registered in this node, $S(f, d)$ is computed to check whether or not $S(f, d) \geq T(f)$ holds. If true, the subscriber specifying f is notified of d. However, we notice that, by "full registration", f is registered in $|f|$ home nodes, which may cause duplicate notifications of d to the subscriber specifying f. For example, in Fig. 2, f_2 is registered in home nodes of three query terms $\{B, D, E\}$. When d is forwarded to three home nodes of $\{B, D, E\}$, these home nodes could duplicately notify the subscriber specifying f_2 of d.

To avoid the duplicate notification above, we introduce the *significant term* $t_{f,d}$ as follows. Given document d and filter f, among those terms which commonly appear in both f and d, we call the term with the highest value of $s(t_i, d)$ the *significant term* (STerm) of f and d, denoted by $t_{f,d}$. Such a significant term makes sense, because its term score $s(t_{f,d}, d)$ contributes *most significantly* to $S(f, d)$. In case of a tie in the highest score, we can break the tie by some rule such as choosing the term that appears the earliest in the document. With the definition of STerm $t_{f,d}$, we give the following notification rule:

> **Notification rule**: Given a qualified filter f, the subscriber specifying f is notified of document d only by the home node of the STerm $t_{f,d}$.

To illustrate this notification rule, we use Fig. 2 as an example, where document d contains the pairs of terms $\{A, B, C, D\}$ and associated term scores. Also for simplicity, we assume that all 3 filters of Fig. 2 are specified with the same threshold 1.0. Consider that document d is forwarded to home nodes of 4 terms $\{A, B, C, D\}$. When d reaches the home node of B, it can be observed that, among 3 query terms in the locally registered filter f_2, term B contributes most significantly to $S(f_2, d)$, and thus it is the significant term $t_{f_2,d}$. Therefore, via the home node of B, the subscriber specifying f_2 is notified of d. Meanwhile, when d reaches the home node of D, by the notification rule, the home node of D will not notify the subscriber specifying f_2 of d, because term D is not a significant term $t_{f_2,d}$. Similarly, terms A and C are significant terms of $t_{f_1,d}$ and $t_{f_3,d}$, respectively. Thus, those subscribers specifying f_1 and f_3 are uniquely notified of d, via home nodes of A and C, respectively.

Based on the notification rule, *no nodes are required to remember the history information of those already-notified documents*. Instead, in Sieve [14], proxy nodes of subscribers have to remember the entire history information and then filter out duplicate publications. Clearly, the cost used to maintain the entire history information is non-trivial.

Before continuing the following sections to present main algorithms, we first use Fig. 3 to highlight their relations. First, we consider the simple data model. Among two selection algorithms (Algorithms 1–2) used to select meaningful terms, Algorithm 1 targets at a simple case that every filter f uses the default threshold T, and f contains at most $|f|_s$ query terms; and Algorithm 2 considers a general scenario that filters have personalized thresholds and different numbers of query terms. Algorithms 1 and 2 are used to select a very small number of meaningful terms (called TTerms) to reduce the publishing amount. Next, the refinement algorithm (i.e., Algorithm 3) reduces the unit-publishing cost by pruning a document d if d is irrelevant to filters registered to the home node of a selected TTerm. This algorithm calls a subfunction related to the document pruning algorithm (given by Algorithm 4). Finally, for the extended data model, Algorithm 5 demonstrates the steps of insertion and

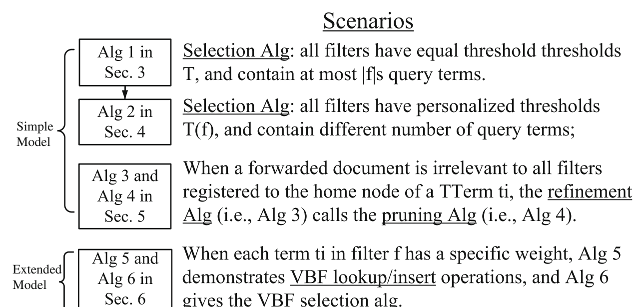


Fig. 3 Relationships of proposed algorithms

lookup operations over a proposed structure, named a value-based bloom filter (in short VBF), which summarizes filter weights. After that, Algorithm 6 extends Algorithm 2 with the help of VBF.

3 Default forwarding

In this section, we assume that every subscriber has an equal (default) threshold $\mathrm{T}(f) = \mathrm{T}$.

3.1 Default selection algorithm

Recall that, only if the condition $\mathrm{S}(f, d) \geq \mathrm{T}$ holds, document d is forwarded to the subscriber specifying f. We show how to use such a condition for document forwarding. If sorting all terms in d in *descending* order of $s(t_i, d)$, we can position the threshold T in the sorted list, as illustrated by the following example. In Fig. 4a, all term scores are sorted in descending order, the sum of term scores from M to E is 0.68, and the sum from M to D is 1.18. Then, the threshold $\mathrm{T} = 1.0$ is positioned between D and E. Since the sum of term scores from M to E is less than $\mathrm{T} = 1.0$, we call the terms from M to E *Below Threshold Terms* (BTerms), and the remaining terms in d (i.e., terms from D to A) *Threshold Terms* (TTerms).

Normally, BTerms and TTerms are defined as follows. Given a sorted list of term scores in document d with $s(t_1, d) \geq \cdots \geq s(t_{|d|}, d)$, for any term t_h with $1 \leq h \leq |d|$, if $\sum_{i=h}^{|d|} s(t_i, d) < \mathrm{T}$ holds, then any term t_i with $h \leq i \leq |d|$ is a BTerm, and term t_i with $1 \leq i < h$ is a TTerm. With BTerms and TTerms, we can find the following results with respect to document d and threshold T.

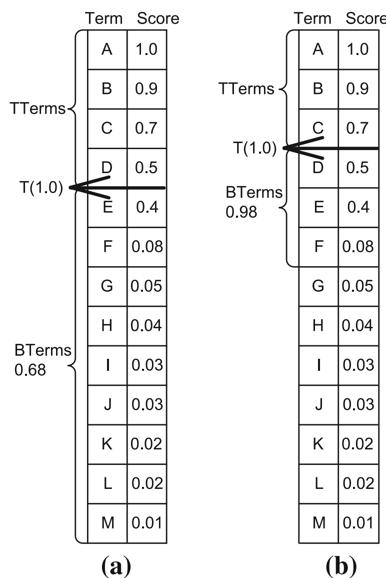


Fig. 4 Default Forwarding: **a** using a default threshold $\mathrm{T} = 1.0$; **b** improved for short filters with $|f|_s = 3$

Claim 1 By the definition of BTerms, the sum of term scores for any subset of BTerms in document d is less than T.

The correctness of Claim 1 is obvious. For example, in Fig. 4a, for the full combination of BTerms from M to E, the sum of term scores, 0.68, is less than the threshold 1.0. Then, for a partial combination for BTerms (e.g., $\{E, F, G\}$), the sum of their term scores is further less than $\mathrm{T} = 1.0$.

Theorem 1 *Given document d with $\mathrm{S}(f, d) \geq \mathrm{T}$, a qualified filter f contains at least one TTerm of d.*

Proof By contradiction: if no TTerm appears in f, f is only composed of BTerms. By Claim 1, we find that the sum of term scores for all query terms in f must be less than T, contradicting the assumption of Theorem 1. Hence, Theorem 1 holds. □

Given a filter $f_2 : \{B, D, E\}$ (in Fig. 2) satisfying $\mathrm{S}(f_2, d) \geq 1.0$, two TTerms $\{B, D\}$ appear in f_2.

By Theorem 1, we can find that a *qualified filter* f contains at least one TTerm, which is selected from document d. Meanwhile, Sect. 2.2 states that filter f is "*fully* registered" in the home node of each term in f. Thus, any term in filter f is either a TTerm of d, or a BTerm of d, or not inside d. Hence, "full registration" guarantees that a qualified filter f satisfying $\mathrm{S}(f, d) \geq \mathrm{T}$ must be registered in the home node of a TTerm of d. Consequently, when document d is published, we can only require that d be forwarded to home nodes of TTerms in d, without causing false dismissals. We call this approach *partial forwarding*.

Compared with the *full-forwarding* approach that forwards d to the home node of each distinct term $t_i \in d$, partial forwarding reduces the forwarding cost. For example, when document d of Fig. 4a is published, d is only forwarded to home nodes of 4 TTerms $\{A, B, C, D\}$. Instead, full-forwarding forwards d to home nodes of 13 terms from A to M.

3.2 Optimization for short filters

Based on real query logs collected from two search engines (See Sect. 9), we find that the major input queries are composed of only several (e.g., 3) terms. In this section, we will show that we can further reduce the forwarding cost by selecting a smaller number of TTerms.

Recall that the definitions of TTerms and BTerms in Sect. 3.1 implicitly assume that a filter is composed of an *arbitrary* number of terms. Now suppose $|f|_s$ is the largest length among all filters. Given $|f|_s$, we redefine TTerms as follows: in the sorted list of term scores $s(t_i, d)$, the sum of $|f|_s$ consecutive term scores should be less than T. Then,

among these $|f|_s$ consecutive term scores, we use the term t_h with the highest one to indicate a term in d is either a TTerm or a BTerm: any term in d having a higher term score than $s(t_h, d)$ is a TTerm; otherwise, it is a BTerm.

For example, in Fig. 4b, $|f|_s$ is equal to 3. Since the sum of 3 term scores related to F, E and D is only 0.98, we determine that any term t_i with $s(t_i, d) \leq 0.5$ is a BTerm; and the remaining terms as TTerms. Compared with Fig. 4a where the sum of term scores starts from the bottom line, now we only require the sum of $|f|_s$ consecutive term scores to be less than T. Obviously, with a smaller value of $|f|_s$, we boost BTerms to a higher position in the sorted list and select a smaller number of TTerms. Thus, forwarding d to home nodes of a smaller number of TTerms consumes less forwarding cost.

Similar to Claim 1 and Theorem 1, we derive the following results for the newly defined BTerm and TTerm:

Claim 2 The sum of term scores by any combination of $|f|_s$ BTerms in document d is less than T.

Theorem 2 *Given a document d with $\mathrm{S}(f, d) \geq \mathrm{T}$, a qualified filter f composed of at most $|f|_s$ terms contains at least one TTerm of d.*

Algorithm 1 lists the pseudocode to utilize short filters to select TTerms. During the loop from step 8 to step 13, S is re-initiated to aggregate at most $|f|_s$ term score. If $\mathrm{S} \geq \mathrm{T}$ is met, the remaining terms from t_1 to t_j are returned as TTerms.

Algorithm 1 Default_Selection (threshold T, doc d, size $|f|_s$)

```
Require: |d| term scores s(t_i, d) of document d are reverse sorted with s(t_1, d) ≥
    ··· ≥ s(t_|d|, d);
1: for i = |d| to 1 do
2:   S ← 0;
3:   if i − |f|_s > 0 then
4:     jUpper ← i − |f|_s;
5:   else
6:     jUpper ← 1;
7:   end if
8:   for j = i to jUpper do
9:     S ← S + s(t_j)
10:     if S ≥ T then
11:       return t_1, ...t_j as selected TTerms;
12:     end if
13:   end for
14: end for
15: return null;
```

3.3 Summary

Full registration is the key of STAIRS to enable partial forwarding with less forwarding cost. Documents d_1 and d_2, even with the same term membership, may produce different sorted lists of term scores. A term t_i may be a TTerm in d_1, but it may be a BTerm in d_2. Full registration guarantees that d is forwarded only to home nodes of selected TTerms. Moreover, when d reaches home nodes of the selected TTerms, the notification rule given in Sect. 2.4 can avoid maintaining the entire history information of already-notified documents.

4 Adaptive forwarding

So far, Sect. 3 assumes that every filter f is specified with an equal threshold T. In this section, we first show that the default selection algorithm (i.e., Algorithm 1) incurs a high forwarding cost or high false dismissal ratio for filters specified with personalized thresholds. Then, we describe how STAIRS can utilize a filter summary structure to select TTerms.

4.1 Challenges to specify personalized thresholds

As shown in the following scenarios, the default selection algorithm may cause a high forwarding cost or high false dismissal ratio:

- There can be problems when the value of T in Algorithm 1 is either too high or too low compared to a specific personalized threshold. If T is too low, it will produce many unwanted documents since selected TTerms by T are excessive; if too high, false dismissals become another issue.
- Not all terms in d must appear in the registered filters. For example, in Fig. 5a, terms C and E do not appear in 9 filters f_1, ..., f_9. Since E does not appear in these filters, we should skip the term score of E during the TTerm selection. Next, when term E is excluded, the sum of term scores from D to M becomes 0.78, less than $\mathrm{T} = 1.0$. Thus, three remaining terms A, B and C are chosen as TTerms. Similarly, since term C does not appear in 9 filters, we discard C, and only select A and B as TTerms.
- Suppose that we set $|f|_s = 6$ in Algorithm 5. However, among all terms appearing in filter f, some of them do not appear in d. For example, filters f_5 and f_7 in Fig. 5b contain at least 6 terms; however, only 3 terms of f_5 (i.e., G, I, and K) appear in d, and 3 terms of f_7 (i.e., B, H, and J) appear in d. Thus, $|f|_s = 3$, instead of $|f|_s = 6$, is enough to correctly select TTerms without false dismissals.

As a summary, enough filter information (including thresholds, filter sizes, and query terms) is important for a selection algorithm to reduce a high false dismissal ratio and a high forwarding cost. In the following subsections, by using a filter summary structure, we show how the proposed algorithm can correctly select a small number of TTerms to reduce the forwarding cost.

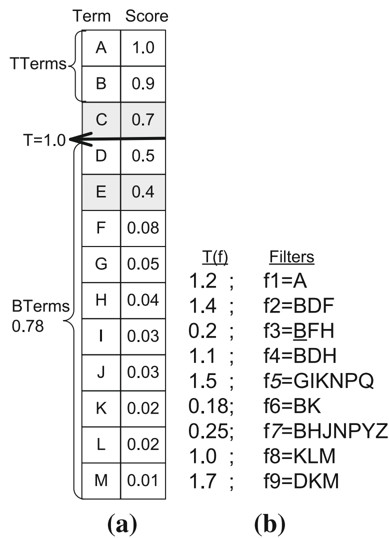


Fig. 5 Adaptive Forwarding: **a** Issues of Default Selection Algorithm; **b** Personalized Filters.

4.2 Selecting TTerms by HiBloom

We use a hybrid structure of histogram and bloom filters, named HiBloom, to summarize filters. The histogram is used to capture filter thresholds, and bloom filters are used to encode query terms. For simplicity, we only consider equi-width histograms, but our approach could be easily generalized to more sophisticated histogram variants [15] (at higher run-time costs, however). Given b buckets, the whole range of filter thresholds are divided into b even intervals. Suppose the k-th bucket ($1 \le k \le b$) is associated with a range of $(lb_k, ub_k]$ where lb_k and ub_k are the lower/upper bound of this bucket. For any filter f with a threshold inside $(lb_k, ub_k]$, query terms in f are encoded by a compact synopsis, represented by a bloom filter, denoted by bf_k; moreover, among all filters inside such a bucket, we maintain the minimal threshold, denoted by T_k. Therefore, each bucket is uniquely associated with a bloom filter bf_k and a threshold T_k.

Note that the HiBloom structure is similar to Histogram-Blooms [19]. However, HiBloom is used for summarizing filters, whereas HistogramBlooms are for documents. Moreover, Sect. 4.3 will improve the precision of HiBloom to summarize queries by tuning the number of bits used by a bloom filter, and Sect. 7 will extend HiBloom to summarize queries associated with term weights.

After receiving the request to publish document d, a peer node in the DHT selects TTerms via HiBloom, shown as follows. For each bloom filter associated with a minimal threshold T_k, we follow Algorithm 2 to select TTerms by the filter size $|f|_s$ (we consider the lower bound T_k as the threshold). In the process of selecting TTerms, we check whether or not term $t_i \in d$ also appears in the bloom filter. If the answer is true, $s(t_i, d)$ contributes to $\mathrm{S}(f, d)$. Otherwise, we directly consider this term as a BTerm. With HiBloom, the selection algorithm can select a small number of TTerms and achieve a low forwarding cost. Moreover, the term membership checking of bloom filters does not produce false negatives.

Algorithm 2 Selection_BF (bloom filter bf_k, threshold T_k, size $|f|_s$, document d)

Require: $|d|$ term scores $s(t_i, d)$ of document d are decreasingly sorted with $s(t_1, d) \ge \ldots \ge s(t_{|d|}, d)$;
1: $d' \leftarrow$ null;
2: **for** $i = 1$ to $|d|$ **do**
3:   **if** $bf_k.lookup(t_i)$ is true **then**
4:     add $\langle t_i, s(t_i, d)\rangle$ to d'
5:   **end if**
6: **end for**
7: **return** Default_Selection(T_k, $|f|_s$, d');

Algorithm 2 shows the pseudocode to select TTerms. First, lines 1–6 initiate a document d' with null, and then add $\langle t_i, s(t_i, d)\rangle$ to d', where $t_i \in d$ also appears in the bloom filter. Since $s(t_i, d)$ is in decreasing order inside the original document d, $s(t_i, d')$ is also decreasingly sorted in d'. After that, we call the function Default_Selection in Algorithm 1 to select TTerms.

4.3 Maintaining HiBloom structure

Due to a large number of filters to be summarized, the precision of HiBloom is important to reduce the false positive ratio. First, an easy approach to improve the precision is to increase the number of buckets in HiBloom, but incurring a higher space cost. In this subsection, we mainly focus on how to improve bloom filters used by HiBloom.

Suppose each bloom filter in the HiBloom structure uses k hash functions and a vector of m bits. For n terms summarized by the bloom filter, we have the following Eq. [13], involving m, n and k, to minimize the false positive ratio:

$$(1/2)^k \approx (0.6185)^{m/n} \qquad (3)$$

Based on Eq. 3, we can find an optimal number n that minimizes the false positive ratio produced by a bloom filter. With b buckets, the HiBloom structure can summarize a total number of $n * b$ terms.

To improve HiBloom, we give the following observation: given *lower* thresholds T_k, Algorithm 2 selects *more* TTerms, and vice versa. On the other hand, HiBloom contains b buckets. Each bucket is associated with a specific threshold T_k. Thus, if each bloom filter is assigned with the same number, m, of bits, due to the false positive issue of bloom filters, those bloom filters associated with *lower* thresholds could *falsely* select *more* TTerms than those with *higher* thresholds. Therefore, the ratio of falsely selected TTerms by the whole HiBloom structure is larger than the standard false positive ratio [13].

From the above observation, instead of assigning an equal number of bits to every bloom filter of HiBloom, for the

bloom filter associated with a *lower* threshold T_k, we assign it with a *larger* number of bits, and vice versa. Consequently, by this biased assignment of the number of bits, the *overall false positive ratio* of the whole HiBloom structure can be smaller than the standard false positive ratio.

Our experiment shows that using the biased assignment of bits can spend less space cost to achieve the same false positive ratio as the standard bloom filter. In the experiment, we implement four kinds of bloom filters respectively with $m = 2^{20}, 2^{16}, 2^{12}$ and 2^8. All b buckets of HiBloom are divided into 4 groups by the bucket sequence ID: $[1, b * 0.05]$, $(b * 0.05, b * 0.2]$, $(b * 0.2, b * 0.5]$, and $(b * 0.5, b]$. For the first group of buckets associated with the smallest thresholds, we assign the largest number, $m = 2^{20}$ of bits, for the second one, $m = 2^{16}$ of bits, for the third one, $m = 2^{12}$ of bits, and the last group of buckets with the largest thresholds, we assign it with the least number, $m = 2^8$ of bits.

To build a global HiBloom in a DHT-based P2P network, we leverage the scalable aggregation tree [46] to summarize registered filters. Each node in the aggregation tree propagates the local HiBloom up to its parent. The parent merges received HiBlooms, together with the local one. That is, inside the same buckets of multiple received HiBlooms, bloom filters are merged by the union operation. Next, the merged HiBloom continues the propagation until the root builds the global HiBloom. Finally, the global HiBloom is propagated down across the whole aggregation tree until each node receives this global HiBloom.

5 Document refinement

Sections 3 and 4 have presented algorithms to reduce the number of publications. In this section, we present a scheme to reduce the unit-publishing cost.

After term t_i is selected as a TTerm, document d is forwarded to the home node of t_i. However, there still exists the case where $S(f, d) < T(f)$ holds for *every* filter f that is locally registered in the home node of t_i. In this case, forwarding d to this home node is unnecessary.

Based on the observation above, we need to refine the document forwarding. That is, along the forwarding path toward the home node of t_i, document d, if irrelevant to all filters registered in the home node of t_i, should be discarded *before* its arrival at the destination. Clearly, the document refinement can reduce the unit-publishing cost.

5.1 Refinement structure

To enable the document refinement, we first introduce a lightweighted refinement structure, by which an irrelevant document can be pruned. For each selected TTerm t_i, there is an associated refinement structure, denoted by $\mathcal{R}_{t_i}$. The refinement structure $\mathcal{R}_{t_i}$, initially built in the home node of t_i, consists of a bloom filter and a lower bound threshold. The bloom filter encodes the query terms of all filters containing t_i. The lower bound threshold is the minimal threshold among all filters containing t_i.

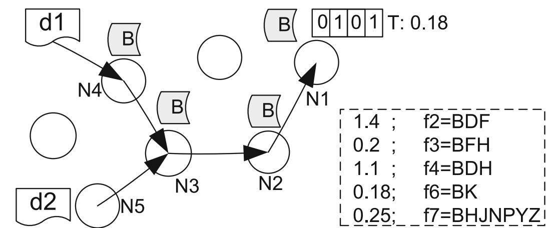


Fig. 6 An example of using the document refinement structure $\mathcal{R}_B$ in N_3 to prune an irrelevant document d_2

In Fig. 6, N_1 is the home node of term B, and registers 5 filters containing term B (i.e. f_2, f_3, f_4, f_6 and f_7 of Fig. 5). The refinement structure of term B, $\mathcal{R}_B$, consists of the bloom filter encoding all query terms in these filters and the lower bound threshold 0.18.

5.2 Refinement scheme

Given $\mathcal{R}_{t_i}$, the refinement scheme is shown as follows. In each intermediately visited node toward the home node of t_i, an incoming document d checks whether or not $\mathcal{R}_{t_i}$ is locally available. If none intermediate node has locally maintained $\mathcal{R}_{t_i}$, d finally reaches the destination (i.e., the home node of t_i). Then, document d is matched with all locally registered filters. If the condition $S(f, d) \geq T(f)$ holds, following the proposed notification rule, the subscriber specifying f is notified of d. Meanwhile, since $\mathcal{R}_{t_i}$ is unavailable in all previously visited intermediate nodes, the home node of t_i copies $\mathcal{R}_{t_i}$ to these intermediate nodes.

Next, another document d' toward the home node of t_i may meet an immediate node having $\mathcal{R}_{t_i}$. Via $\mathcal{R}_{t_i}$, the document pruning algorithm in Sect. 5.3 determines whether or not d' continues its remaining forwarding. If d' is not pruned by $\mathcal{R}_{t_i}$, d' continues the remaining forwarding. Otherwise, d' is discarded. Similar as before, $\mathcal{R}_{t_i}$ will be copied to previously visited intermediate nodes, which do not locally maintain $\mathcal{R}_{t_i}$.

We use Fig. 6 to illustrate the scheme above. First, d_1 is forwarded to the home node of term B, i.e., N_1. Since no intermediate node maintains $\mathcal{R}_B$, d_1 finally reaches N_1, which then copies the local $\mathcal{R}_B$ to 3 intermediate nodes (N_2, N_3, and N_4). Next, d_2 meets $\mathcal{R}_B$ at N_3. Then, N_3 determines whether or not d_2 is pruned.

Algorithm 3 gives the pseudocode of the refinement scheme. The publication message *msg* contains three parts: (i) document d; (ii) TTerm t_i (so that *msg* is forwarded to the home node of t_i), and (iii) a list of previously visited node

Algorithm 3 REFINE_SCHEME(publication message msg)

```
1: if the local refinement structure R_{t_i} is unavailable then
2:    add local_ip to msg.lnodes;
3:    return true;
4: end if
5: let isPruned = DOC_PRUNE(R_{t_i}, msg.doc, msg.t_i);
6: if isPruned == false then
7:    return true;
8: else
9:    send the local R_{t_i} to each member in msg.lnodes;
10:   return false;
11: end if
```

Algorithm 4 DOC_PRUNE (refinement structure $\mathcal{R}_{t_i}$,doc d, TTerm t_i, filter size $|f|_s$)

```
Require: s(t_1, d) ≥ ··· ≥ s(t_{|d|}, d);
1: m ⇐ 0; S ⇐ 0.0;
2: for j = 1 to |d| do
3:    if (the document term t_j ∈ d is the just the TTerm t_i) then
4:       m ⇐ 1; S ⇐ s(t_j, d);
5:       for k = j + 1 to |d| do
6:          if (term t_k ∈ d is found in bf) then
7:             m ⇐ m + 1; S ⇐ S + s(t_k, d);
8:             if (m == |f|_s) then breaks both for loops and jumps to line 14;
9:          end if
10:      end for
11:   end if
12: end for
13:
14: if S < R_{t_i}.lb then
15:    return true;
16: else
17:    return false;
18: end if
```

addresses, denoted by $lnodes$. Along the document forwarding path, each intermediate node processes msg by a document pruning algorithm, which will be given by Sect. 5.3.

As shown in Algorithm 3, if $\mathcal{R}_{t_i}$ is not available in an intermediate node, the local address is added to $lnodes$ of msg, and msg continues its forwarding (lines 2–3).

If $\mathcal{R}_{t_i}$ is locally available, the intermediate node conducts the document *pruning* operation (line 5). If not pruned, msg continues its forwarding (line 7). Otherwise, if msg is pruned, given $lnodes$ of msg, the local $\mathcal{R}_{t_i}$ is copied and sent back to each member in $lnodes$ (line 9). Since msg is pruned, the forwarding is terminated (line 10).

There are two properties pertaining to Algorithm 3:

- If msg is pruned, the remaining forwarding is terminated before its arrival at the home node of t_i. Therefore, the routing hop count of msg is smaller than the original hop count $O(\log N)$. Section 6 will analyze the performance of Algorithm 3.
- Only when msg is not pruned, the intermediate node adds its local address to $msg.lnodes$ and expects $\mathcal{R}_{t_i}$ from the home node which can prune msg successfully. Since the number of nodes in $lnodes$ is at most $O(\log N)$ and the data size of $\mathcal{R}_{t_i}$ is significantly smaller than that of d, the communication cost consumed by sending $\mathcal{R}_{t_i}$ (line 9) is trivial.

5.3 Pruning algorithm

In this section, we show the details of the pruning algorithm, which is used by Algorithm 3. The pseudocode of the pruning algorithm is given by Algorithm 4.

We first highlight the basic idea. To enable this pruning algorithm, we ensure that term scores in document d are sorted in descending order with $s(t_1, d) \geq \cdots \geq s(t_{|d|}, d)$. Similar to previous selection algorithms, we compute the sum of $|f|_s$ term scores for those terms commonly appearing in both d and the bloom filter (lines 2–12). If the sum S is *smaller* than the minimal threshold of $\mathcal{R}_{t_i}$, i.e., $\mathcal{R}_{t_i}.lb$ (in line 14), we safely discard document d without continuing the remaining forwarding (line 15). Otherwise, d continues its remaining forwarding (line 17).

Note that though Algorithm 4 processes terms in d in a top-down manner from t_1 to $t_{|d|}$, lines 3–11 compute the sum of term scores starting from the TTerm t_i, instead of t_1. It is because along the forwarding path to the home node of TTerm t_i, every qualified filter registered in this home node must contain t_i. Therefore, we treat t_i as the *significant term* of those filters and calculate $S(f, d)$ by the sum of term scores from $s(t_i, d)$ to other smaller term scores $s(t_k, d)$ (lines 5–10). Clearly, the fact that t_i is the significant term can help precisely approximate $S(f, d)$.

5.4 Maintenance

Consider new filters are registered to STAIRS and existing ones are deregistered. The maintenance of filters requires that $\mathcal{R}_{t_i}$ in those intermediate nodes be consistent with the one in the home node of t_i. To support this requirement, an *overlay tree* is maintained to connect those nodes having $\mathcal{R}_{t_i}$. In detail, the home node of t_i is the root of this tree, and all intermediate nodes having $\mathcal{R}_{t_i}$ form subtrees of the root. For example in Fig. 6, the overlay tree is rooted at node N_1. After $\mathcal{R}_{t_i}$ is copied to node N_5, N_5 will join the tree, and the whole tree contains 5 nodes ($N_1 ... N_5$). This tree structure is similar to the DHT prefix tree [24].

The cost to maintain the tree structure is low because all edges, based on existing connections in the DHT, have been optimized (e.g., by the network proximity [32]). Then, if $\mathcal{R}_{t_i}$ in the home node of term t_i is updated, this updated $\mathcal{R}_{t_i}$ is propagated across the whole tree. This propagation can be further optimized by the multicast message [33].

6 Analysis

In this section, we analyze the cost of previously proposed algorithms.

6.1 Analysis of selection and pruning algorithms

The key of a selection algorithm is to approximate $\mathrm{S}(f, d)$ with the help of various filter information. For example, via the filter size $|f|_s$, Algorithm 1 approximates $\mathrm{S}(f, d)$ by the sum of at most $|f|_s$ term scores. Via HiBloom, Algorithm 2 approximates $\mathrm{S}(f, d)$ by the sum of at most $|f|_s$ term scores of those terms appearing in both d and bloom filters of HiBloom. Besides, the pruning algorithm (Algorithm 4) approximates $\mathrm{S}(f, d)$ via the refinement structure $\mathcal{R}_{t_i}$.

Clearly, the goodness of the proposed algorithms depends upon how $\mathrm{S}(f, d)$ is precisely approximated. Therefore, we are interested in approximation bounds. The analysis is given as follows.

For the two selection algorithms, Algorithm 2 is the general one by considering the personalized thresholds. Thus, we focus on the analysis of Algorithm 2.

Theorem 3 *The approximated value of* $\mathrm{S}(f, d)$ *in Algorithm 2 is at most* $|f|_s$ *times of its real value, and at least* $1/|f|_s$ *of its real value.*

Proof Algorithm 2 only processes those terms *commonly* appearing in both d and bloom filters (by the term membership checking in line 6 of Algorithm 2). It means that after line 3, filter f contains at least one term t_i commonly appearing in both d and bloom filters (with a false positive ratio). After that, $\mathrm{S}(f, d)$ is approximated by the sum of $s(t_i, d)$ and at most $(|f|_s - 1)$ remaining term scores (lines 8–13 of Algorithm 1).

Because term scores in d are sorted in descending order, $s(t_i, d)$ is no smaller than the remaining $(|f|_s - 1)$ terms cores. Then, in the best case where all of the $(|f|_s - 1)$ remaining terms appear in both d and f, the approximate value of $\mathrm{S}(f, d)$ is just equal to its real value. In the worst case where only one term t_i (except the $(|f|_s - 1)$ remaining terms) commonly appears in both d and f, the real value of $\mathrm{S}(f, d)$ is $s(t_i, d)$, and Algorithm 2 approximates $\mathrm{S}(f, d)$ by the sum of $s(t_i, d)$ and the $(|f|_s - 1)$ remaining term scores. Due to the descending order of term scores in d, $s(t_i, d)$ must be no smaller than the $(|f|_s - 1)$ remaining term scores. Thus, the approximated value of $\mathrm{S}(f, d)$ is at most $|f|_s$ times of its real value. Meanwhile, considering the case where all $|f|_s$ term scores are equal, the approximated value of $\mathrm{S}(f, d)$ (no smaller than $s(t_i, d)$), is at least $1/|f|_s$ of its real value. □

For all filters having $|f| = 1$, Algorithm 2 optimally approximates $\mathrm{S}(f, d)$ just equal to its real value. However, a large $|f|$ produces a loose approximation of $\mathrm{S}(f, d)$. Real data sets in our experiment show that end users prefer *short* queries, on average with $|f| \approx 3.0$. Those short queries obviously favor our algorithm with a tight approximation of $\mathrm{S}(f, d)$.

The pruning algorithm, i.e., Algorithm 4, also follows Theorem 3. However, it can more tightly approximate $\mathrm{S}(f, d)$ than Algorithm 2, because $\mathcal{R}_{t_i}$ only encodes query terms in those filters containing t_i.

6.2 Analysis of refinement scheme

In this section, we analyze the refinement algorithm (Algorithm 3). We assume that, among D documents, totally M terms are selected as TTerms. Among these M terms, some terms are selected as TTerms more frequently than others. Hence, among M terms, there could be only I *distinct* terms with $0 \le I \le M$. For $1 \le i \le I$, term t_i is selected as a TTerm by a ratio r_i with $\sum_{i=1}^{I} r_i = 1$. That is, among M documents, t_i is selected as a TTerm from $r_i \cdot M$ documents. Intuitively, r_i indicates the *popularity* of t_i in terms of how t_i is frequently selected as a TTerm.

For the sake of fairness, we assume all publication sources are *randomly* distributed across the DHT. Thus, the publication sources of those $r_i \cdot M$ documents can form the overlay tree (given in Sect. 5.4). The height of such an overlay tree is at least $\log(r_i \cdot M)$ (by treating those $\log(r_i \cdot M)$ publication sources as intermediate nodes of the overlay tree). If more other nodes join the overlay tree, the height is larger than $\log(r_i \cdot M)$.

Along the forwarding path to the home node of t_i, the refinement structure $\mathcal{R}_{t_i}$ in each intermediate node might prune document d successfully. Suppose d is successfully pruned by $\mathcal{R}_{t_i}$ with probability ρ_i, the average hop count of forwarding message *msg* to the home node of t_i, denoted by H_i, is:

$$H_i = \log N - \rho_i \cdot \log(r_i \cdot M). \tag{4}$$

Given I distinct TTerms, the average hop count is:

$$H = \sum_{i=1}^{I} r_i \cdot H_i = \sum_{i=1}^{I} r_i \cdot \left[\log N - \rho_i \cdot \log(r_i \cdot M)\right]. \tag{5}$$

Due to $\sum_{i=1}^{I} r_i = 1$, the equation above becomes:

$$H = \log N - \sum_{i=1}^{I} \left[\rho_i r_i \log(r_i \cdot M)\right]. \tag{6}$$

For Eq. 6, we give the following discussions:

- The first subitem $\log N$ is related to the original hop count offered by the DHT. The reduction of the hop count H is mainly decided by parameters r_i and ρ_i (we assume that both M and N are given variables).
- In the second subitem $\sum_{i=1}^{I} \left[\rho_i r_i \log(r_i \cdot M)\right]$, a higher punning ratio ρ_i leads to a smaller hop count H (the

value of ρ_i directly depends on how the pruning algorithm approximates $S(f, d)$, which has been analyzed in Sect. 6.1).

– If ρ_i is removed, the second subitem above becomes $\sum_{i=1}^{I}\left[r_i \log\left(r_i \cdot M\right)\right]$, which can be further transformed to $\sum_{i=1}^{I}\left[r_i \log\left(r_i\right)\right] + \log M$. The value of $\sum_{i=1}^{I}\left[r_i \log\left(r_i\right)\right]$, i.e., the entropy of r_i, depends on the distribution of r_i. A skewed distribution of r_i leads to a large value of this subitem, correspondingly a small value of H. It makes sense because given a skewed distribution of r_i, a large number of $\mathcal{R}_{t_i}$ are propagated across intermediate nodes. These intermediate nodes then have more chances to prune d.

7 Extended data model

In this section, we adapt the aforementioned data structures and algorithms to the extended data model, where term $t_i \in f$ is associated with a normalized weight $nw(t_i, f)$.

7.1 Adapting selection algorithms

Among two selection algorithms (Algorithms 1 and 2), Algorithm 2 generally considers the personalized thresholds. Thus, we focus on adapting Algorithm 2 to the extended data model, and the proposed technique can be similarly adapted to Algorithm 1.

7.1.1 Adapting HiBloom to extended model

Recall that in Sect. 4, HiBloom uses the standard bloom filter to encode query terms. However, HiBloom does not cover the weight information. In particular, each term t_i in f is associated with a normalized weight $nw(t_i, f)$. Moreover, given the same term t_i, the value of $nw(t_i, f)$ varies with respect to filters.

We improve HiBloom to summarize the filter information containing the weight information. Instead of standard bloom filters, we utilize the *counter bloom filter* [6] (in short CBF), where each entry in the bit vector is a counter (e.g., consisting of 4 bits to represent the counter). Note that the entry of the standard bloom filter is a single bit, either 0 or 1.

Following the idea of CBF, we propose a *value-based bloom filter* (in short VBF) as follows. To summarize both query terms and normalized weights, each entry in VBF is associated with a weight value. Due to the k hash functions in VBF, t_i is mapped to r entries. After query terms and associated weights are summarized by VBF, the entries of t_i might be reset with different weights, called a *hash collusion*. When a hash collusion occurs, a smaller weight replaces the original one, called the *minor* policy. The rationale of this policy is that, given a selection algorithm, the minor threshold selects more TTerms than the bigger one. Otherwise, if using a bigger threshold, the selection algorithm will falsely miss the TTerms that are required by the minor one.

After filters are summarized to a VBF by insert operations, a lookup of term t_i returns the *biggest* weight among r entries associated with t_i. Returning the *biggest* weight is essentially consistent with the minor policy. Algorithm 5 shows the pseudocodes for insert and lookup operations, respectively. Line 4 of the insert operation is related to the minor policy, and line 6 of the lookup operation finds the biggest weight among r entries associated with t_i.

Algorithm 5 Operations of Value_Bloom_Filter

```
1: INSERT (term t, weight value v);
2: /* there are k hash functions */
3: for i = 1 to k do
4:   if entry(hash_i(t)) is null || entry(hash_i(t)) > v then
5:     entry(hash_i(t)) = v;
6:   end if
7: end for
```

```
1: LOOKUP (term t);
2: v ← 0.0
3: for i = 1 to k do
4:   if entry(hash_i(t)) is null then
5:     return null;
6:   else if entry(hash_i(t)) > v then
7:     v = entry(hash_i(t));
8:   end if
9: end for
10: return v;
```

We use Fig. 7a to illustrate insert and lookup operations of VBF. Among three filters, term A in f_1 has $nw(A, f_1) = 1.0$; term B in f_2 has $nw(B, f_2) = 0.2$. The weights of other terms are also shown in this figure. To summarize these filters, we maintain a VBF with $k = 2$ hash functions and a vector with $m = 10$ bits. We use term A as an example. By $k = 2$ hash functions, term A is mapped to 2 entries respectively with IDs 0 and 3, and $nw(A, f_1) = 1.0$ is stored in both entries. Due to the hash collusion, term D is also mapped to

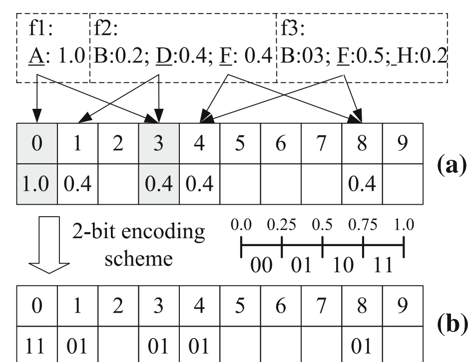


Fig. 7 VBF and s-bit Encoding scheme

the entry with ID 3. By the minor policy, the entry with ID 3 is reset by the minor one, $nw(D, f_2) = 0.4$. To lookup the weight of A, among two weights (i.e., 1.0 and 0.4) in the associated entries with IDs 0 and 3, the *bigger* weight 1.0 is returned. For term A, if and only if the entries of IDs 0 and 3 are both reset by minor weights (due to the hash collusion), the lookup operation falsely returns a value that is smaller than its real weight. Note that for term F appearing in both f_2 and f_3, $nw(F, f_3) = 0.5$ in the entries with IDs 4 and 8 is reset by the minor $nw(F, f_2) = 0.4$. It differs from the aforementioned case caused by the hash collusion.

Though the solution above works, an issue is the high space cost caused by directly storing weights in entries of the VBF. For example, considering the data size of a numeric weight (for example, a float data type typically needs 4 bytes), the overall space of VBF, e.g., having 2^{10} entries, is non-trivial.

To save the space cost, we approximate the exact value of $nw(t_i, f)$ by using an encoding approach and then store the encoded bits into VBF since the normalized value $nw(t_i, f)$ is always inside (0.0,1.0]. In detail, the whole range (0.0, 1.0] can be divided into 2^b (even) intervals. The width of each divided interval is $1.0/2^b$. Then, given an exact value of $nw(t_i, f)$, we can locate the interval I with $lb_I < nw(t_i, f) \leq ub_I$, where lb_I and ub_I are the lower/upper bound of I, respectively. For each of 2^s intervals, we encode the interval by s bits. That is, the first interval with the range of $(0.0, 1.0/2^s]$ is encoded by s bits of 0, and the second interval with $(1.0/2^s, 2.0/2^s]$ by the first $(s-1)$ bits of 0 and the last bit of 1. Similarly, the last interval with $((2^s-1)/2^s, 1.0]$ by s bits of 1.

We use Fig. 7b to illustrate the encoding scheme above. Given $s = 2$, the range (0.0, 1.0] is divided into $2^s = 4$ intervals, where interval (0.0, 0.25] is represented by 00, and interval [0.75, 1.0] is represented by 11. Then, the weight 1.0 in the entry of ID 0 is replaced by 11, and the weight 0.4 is replaced by 01, etc.

The encoding approach above approximates the exact value of $nw(t_i, f)$ by s encoded bits. The approximation precision depends on two factors:

- The number of bits s: when the number s grows, the approximation precision increases. However, it incurs higher space cost.
- The precision of the VBF itself. Recall that in Algorithm 5, the lookup of term $t_i \in f$ might falsely return a value that is *smaller* than $nw(t_i, f)$. This occurs if and only if all of r entries associated with t_i are reset by smaller weights. Obviously, the probability of returning a smaller value is equal to the false positive ratio of the standard bloom filter [6]: $1 - (e^{-kn/m})^k$.

Note that though other approaches to divide intervals (e.g., similar to the equi-depth histogram) can be used, they incur more complicated protocols to merge VBFs (see Sect. 4.3).

7.1.2 *Adapting algorithm 2 to the extended model*

We adapt Algorithm 2 to the extended data model via the proposed VBF. The pseudocode is shown in Algorithm 6. As before, we make sure that only those terms commonly appearing in d and VBF are useful (lines 3–5). If the returned result of VBF is not null (line 4), it indicates that $t_i \in d$ also appears inside the VBF (though with the false positive issue). In line 5, $S(f, d)$ is based on the normalized score function of Eq. 2. After that, Algorithm 6 sorts term scores of d' in descending order (line 8), and then calls function Default_Selection given by Algorithm 1.

Algorithm 6 Select_VBF (value bloom filter vbf, threshold T, size $|f|$, document d)

```
1: d' ← null;
2: for i = 1 to |d| do
3:   nw(t_i) ← vbf.lookup(t_i);
4:   if nw(t_i) is not null then
5:     add ⟨t_i, ns(t_i, d) · nw(t_i)⟩ to d'
6:   end if
7: end for
8: sort term scores in d' with s(t_1, d') ≥ ... ≥ s(t_|d'|, d');
9: return Default_Selection(T, |f|, d');
```

7.2 Extended document pruning algorithms

First, we adapt the refinement structure $\mathcal{R}_{t_i}$ to the extended model. Similar to Sect. 7.1.1, we replace the standard bloom filter in $\mathcal{R}_{t_i}$ by the proposed VBF. In the extended model, no change is needed in the refinement algorithm (i.e., Algorithm 3).

To adapt the pruning algorithm (given in Algorithm 4) to the extended model, we can recompute term scores by $nw(t_i, f) \cdot ns(t_i, d)$ (in lines 4 and 7 of Algorithm 4). Also, in line 6 of Algorithm 4, it is required to decide whether or not the value returned by lookup(t_i) of the VBF is null. If the returned value is not null, we treat that t_i is encoded by the VBF. Except for those above, no other change is needed in Algorithm 4.

8 Related work

We review the literature in the following two areas.

8.1 Distributed information retrieval

In DHT-based distributed IR systems [48,19,39,38,51], document d is indexed in *home nodes* of *all* terms in d, incurring a high maintenance cost. To retrieve interested documents, an end user enters a query f composed of $|f|$ terms, and then the DHT routes f to the home node of *every* term $t_i \in f$. The returned results[1] from $|f|$ home nodes are intersected to determine final results. If we directly adopt this idea to our problem in Sect. 1.2, filter f is first registered to $|f|$ home nodes, then d is forwarded to $|d|$ home nodes, and finally d is matched with f in home nodes of those terms that *commonly* appear in both d and f. Two issues are related to this scheme:

- The cost of forwarding d to $|d|$ home nodes is high, because the number of terms in d, $|d|$, is typically large. For example, one data set used in our experiment contains on average 6,000 terms per document.
- Since f is registered at $|f|$ home nodes, forwarding document d to home nodes of all terms in d will *duplicately* disseminate d to the subscriber specifying f, unless the proposed notification rule is used, or the history of all the disseminated publications is maintained in subscriber proxies so that duplicate documents are avoided [14].

KLEE [19] vertically stores the document information to the home node of each term in the document and uses a thresholding approach to answer distributed top-k queries. Similar to the scheme above, KLEE still spends a high publishing cost of storing the document in home nodes of $|d|$ terms.

pSearch [41], a P2P IR system, utilizes CAN [29] to map latent semantic indexing (LSI) [11] features to peer nodes. After that, queries and documents are mapped into this LSI space, and correspondingly to peer nodes. Considering the dimensionality mismatch between (the high dimensional) LSI and (low dimensional) CAN, pSearch proposes to use multiple CAN instances, mapped to high dimensional LSI space.

In addition, many works have been proposed to study keyword search in unstructured P2P networks. For example, [8, 18,50] focus on the random walk-based search mechanism. BubbleStorm [43] proposes a simple yet efficient probabilistic search approach by reasonably replicating both queries and data objects. All these works are based on probabilistic solutions, with no guarantee to find the expected documents, even if those documents are really stored in P2P networks.

[1] For clarification, the home node of term $t_i \in d$ could only store the pointer or meta-data of a raw document d.

8.2 Information filtering, and Pub/Sub

Those works in Sect. 8.1 are typically related to the pull processing where users *pull* the expected documents. Instead, in the area of information filter (IF) and pub/sub, documents (or events) are *pushed* so that users are notified of the expected documents. Intuitively, pull and push can be treated as the models of query-to-data, and data-to-query (here, data is either raw data, meta-data, or pointer of raw data), respectively.

SIFT [47] is the prominent example of IF. This system considers Usenet News as data sources and collects user profiles in form of queries. In the proposed query index (QI), for every term t_i in a query condition, this query is inserted in the inverted list pointing to t_i. It shares the idea of *full registration*. During the process of document matching, for every term $t_i \in d$, SIFT needs to process the inverted list pointing to t_i. This idea is similar to *full-forwarding*. InRoute [7] is another centralized online filtering system, with help of inference networks to decide whether or not a document matches a query. Different from these works, STAIRS targets at the distributed environments and is based on the idea of *full registration* and *partial-forwarding*.

To overcome RSS scalability issues, FeedTree [34] utilizes P2P networks to cooperatively serve the RSS dissemination. Corona [24] goes a step further via smart distributed polling and client push messages to inform users of new items. As the topic-based pub/sub, FeedTree and Corona use RSS URLs as topics to filter documents. To overcome the high maintenance cost caused by a large number of topics (i.e., RSS feeds), [20] develops a very nice cost model to dynamically cluster RSS feeds. Instead, the proposed fine-grained content filtering model is free from the topic maintenance.

Tryfonopoulos et al. [44] studies the problem of offering a pub/sub functionality on top of structured overlay networks using data models and languages from IR. Different from [44] with the attribute-value model, our work focuses on full-text filtering semantics. Rose et al. [30] utilizes the search engine technology to provide the content filtering function targeting at a cluster environment. Based on the *tf***idf* model, [45] involves the semantic content distribution over a DHT, with some heuristic optimization techniques (e.g., heuristically selecting terms with top term scores, which is consistent with [39]).

Fabret et al. [12], as the classic work of the content-based pub/sub, proposes centralized matching algorithms. The basic idea of these algorithms is to index subscription filters based on attributes with the most selective predicates and then propagate publication events to those indexed subscription filters. Essentially, the idea of [12] is an example of *single registration and full publishing*. Ganguly et al. [14], a distributed content-based pub/sub system, suffers from a high latency problem, due to the counting algorithm used

in subscription clients to wait for the matching results of distributed predicates. Banavar et al. [4], Fabret et al. [12], Ganguly et al. [14] and other content-based pub/sub systems typically focus on the *selectively*-based content filtering. Instead, our work handles the problem of an *aggregation score*-based threshold filtering.

Due to many common techniques, we will compare STAIRS with Ferry [52] in our experiments. As a content-based pub/sub system in a DHT, Ferry utilizes the home node of an attribute as the rendezvous node (RN for short) to match publications with those subscription filters registered to such a RN. Each subscriber in Ferry is associated with a home node. If he/she wishes to subscribe interested events via a filter f, then f is registered to a (actually only *one*) RN, e.g., r. After that, Ferry manages all filters registered to r by an *embedding tree* (EMTree for short) associated with r. Such an embedding tree consists of all intermediate nodes from r to home nodes of those filters registered to r. Intuitively, for each rendezvous node r, there is an associated embedding tree rooted at r, and the 1-hop nodes in the finger table of r are direct children of r. The main purpose of the embedding tree is to move the workload of r to its finger table nodes for load balancing. After an event message is published, this message is first forwarded to *all* rendezvous nodes, which then utilize embedding trees for event delivery. In addition, a RN builds a summary filter to represent filters in the associated embedding tree. Such a summary filter helps filter out useless events. Based on the schema S consisting of a attributes, the number of rendezvous nodes is proportional to a. Many common ideas are shared between STAIRS and Ferry, including using rendezvous nodes based on home nodes of attributes (and home nodes of terms in STAIRS), embedding trees (and overlay trees in STAIRS), and summary filters (and HiBloom in STAIRS).

In addition, the conference version [28] of this paper targets on the simple data model. It mainly focuses on term selection algorithms to reduce the number of messages. Instead, this paper extensively studies both the simple model and the extended model, and proposed the techniques of reducing both the number of messages and the unit-forwarding cost.

As a summary, Fig. 9 compares STAIRS with some (partial) works in the area of content-based pub/sub and distributed IF. First, we consider that STAIRS and distributed IF differ from content-based pub/sub in the form of content: most of content-based pub/sub focus on *structured* information, which follows predefined schemas; while STAIRS and distributed IF (e.g., SIFT [47], etc.) target at unstructured web articles (or summary articles like RSS feeds, or pre-processed with if*idf scores). Next, due to the content of different forms, filters in the content-based pub/sub systems typically consist of selective predicates, while distributed IF either adopts a simple topic-based filtering model (e.g., FeedTree [34], Corona [24], [20], etc.) or full-text like filtering semantics (e.g., STAIRS, SIFT [47], etc.). Finally, for the matching approach, most content-based pub/sub systems explore filter relationships (like covering, overlapping, etc.) to build various kinds of filter indexing structures. Some IF solutions (e.g., SIFT [47]) build IR-like indexes (for example, inverted lists), and some full-text distributed IF solutions [40] directly map a document to the whole semantic space (e.g., LSI). Instead, STAIRS focuses on carefully selecting those terms (or key features) which can contribute significantly to $S(f, d)$. Finally, from the overall policy, the content-based pub/sub [12] adopts the sing-registration and full-matching policy, distributed IF (e.g., SIFT) adopts the full-registration and full-matching policy. Instead, STAIRS is based on the full-registration and partial-forwarding policy.

9 Experiments

In this section, we first show the methodology (Sect. 9.1), and extensively study the performance of the proposed algorithms (Sects. 9.2– 9.5), and compare STAIRS with Ferry (Sect. 9.6). Finally, we summarize experimental results and give discussions (Sect. 9.7).

9.1 Methodology

We evaluate STAIRS in a DHT environment by using FreePastry (http://www.freepastry.org) as the simulator with a default 10,000 nodes, where filter f is registered to home nodes of $|f|$ query terms, and document d is forwarded to the home node of each selected TTerm.

(1) Subscription filters: Though Google Alerts and Microsoft Live Alters provide input interfaces, by which end users subscribe to favorable documents via input keywords, there is no publicly available real data set about using keywords as subscription filters. Instead, we use real data sets of traditional query logs as subscription filters. These logs truly show behaviors of end users to use keywords. We use two real query logs. The first trace log is an 81.3 MB input query history file collected within four months from a popular commercial search engine ("commercial SE" in short), which is quite representative for behaviors of end users in the real world. The

Table 1 Statistics of two query logs

Query parameter	Search.com	Commercial SE
Number of queries	81,497	4,000,000
Avg. num. of terms per query	2.085	2.843
Max. num. of terms per query	11	29
Min. num. of terms per query	1	1

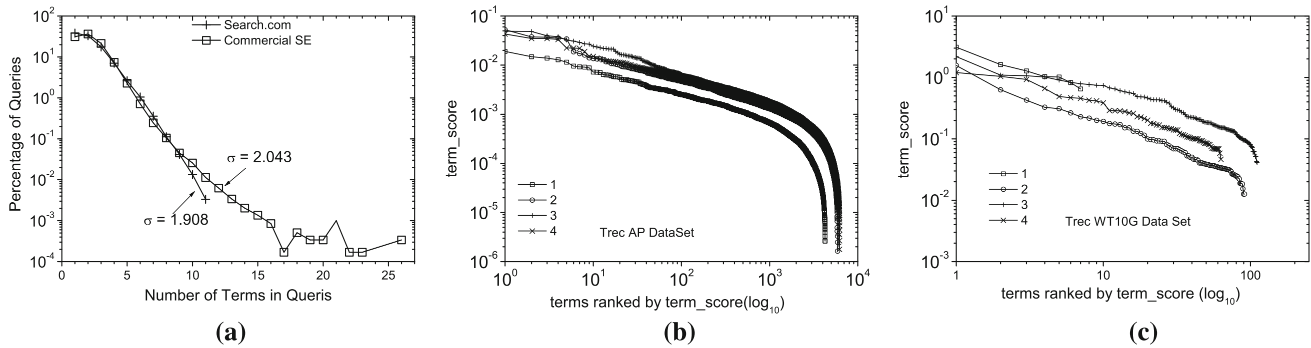


Fig. 8 **a** Two Query Logs consisting of short terms; **b** TREC AP with large articles; **c** TREC WT10G with short articles

second trace is a similar query history file from www.search.com with size 1.27 MB. Table 1 summaries parameters of both query logs. On average, the number of terms per query is 2.085 in search.com, and 2.843 in the commercial SE; the largest number of terms per query in search.com is 11, and that number in the commercial SE is 29. Furthermore, Fig. 8a plots the number of terms in two query logs, where the *x*-axis represents the number of terms per query, and the *y*-axis represents the percentage of queries with the corresponding number of terms. For example, for the search.com trace, the percentages of filters containing 1, 2, 3, and 4 terms are 38.13, 33.09, 17.56, and 7.05, respectively; for the commercial SE trace, the percentages of filters containing 1, 2, 3 and 4 terms are 31.33, 36.42, 17.56, and 7.39, respectively. For both data sets, the standard deviation values over the overall average filter size are 1.908 and 2.043, respectively. Both deviation values indicate that most queries contain around 2–3 terms. Clearly, Table 1 and Fig. 8a indicate that input queries in both trace files are typically composed of only several terms. Hence, if these input queries are used as subscription filters, there are opportunities for our solution to save the forwarding cost, meanwhile achieving fewer false dismissals.

(2) Content document: We use two data sets:

– one based on Text Retrieval Conference (TREC) WT10G web corpus: a large test set widely used in web retrieval research. The data set contains around 10 Gigabyte, 1.69 million Web page documents and a set of queries (we mean the "title" field of a TREC topic as a query). The WT10g data was divided into 11,680 collections based on document URLs. Each collection on average has 144 documents with the smallest one having only 5 documents. The average size of each document is 5.91 KB. The data set was stemmed with the Porter algorithm and common stop words such as "the", "and", etc. were removed from the data set.
– one based on TREC AP: a text categorization task based on the Associated Press articles used in the NIST TREC evaluations. Compared with TREC WT10G data set, the TRACE AP data set is composed of fewer (only 1,050) articles but with a larger number of terms, on average 6054.9 per article. Table 2 summarizes statistics for the test data sets.

	Content-Based Pub/Sub	IF	STAIRS
Content	<attribute, value> Structured Information, typically following predefined schema consisting tens of attributes)	A Web article (either associated with URL or preprocessed pairs of <term, weight>)	<term, weight> Unstructured information (no schema). A single Web doc may include thousands of terms, and all web docs together contain millions of terms.
Filtering semantics	selection-based predicate: STOCK_NAME = 'IBM' STOCK_PRICEL > 120	Either URL or full-text filtering semantics	An aggregation formula, where each term may contribute to TotalScore, which is required to be larger than a threshold.
Matching Scheme	Exploits the relationship between filters such as covering, overlapping etc.	Centralized or Distributed	Carefully select significant terms with the highest contribution to TotalScore(f,d).
Overall Policy	Single-registration Full-Matching	Single-registration Full-Matching	Full-registration Partial-Forwarding

Fig. 9 Comparison of Existing Works and STAIRS

By formula $s(t_i, d) = \frac{freq_{i,d}}{Max_l(freq_{l,d})} \cdot \log \frac{D}{D_i}$, we compute the score of each term in both data sets. In this formula, $\frac{freq_{i,d}}{Max_l(freq_{l,d})}$ represents the value of term frequencies (tf), and $\log \frac{D}{D_i}$ the inverse document frequencies (idf). In detail, $freq_{i,d}$ is the frequency of term t_i in document d; $Max_l(freq_{l,d})$ is the maximal term frequency in d; D_i is the number of documents containing t_i across the whole data set; and D is the total number of documents.

Table 2 Statistics of the TREC data sets

Document parameter	TREC WT10G	TREC AP
Total num. of docs	1,692,096	1,050
Avg. num. of terms per doc	64.808	6,054.9
Max. num. of terms per doc	331	7,320
Min. num. of terms per doc	2	1,303
Total num. of terms	16,382	48,788
Avg. term frequency	130.31	619.48
Max. term frequency	210,089	1,050
Min. term frequency	1	1

Though our experiments pre-compute $s(t_i, d)$ by the approach above, it has shown that in DHT-based distributed IR scenarios it is enough to have an approximation of idf values [9]. By a set of randomly chosen peers to collect and merge statistics, [42] creates an approximation of a global idf. Though it is an open problem to approximate idf in pub/sub scenario [44], one possible solution is to use the approximate solution [42] to compute idf. After the value of idf is ready, we then compute $s(t_i, d)$.

We plot term scores of 4 randomly chosen documents in TREC AP and TREC WT10G in Figs. 8b and c, respectively. From both figures, the term score of TREC WT10G is relatively larger than that of TREC AP due to the effect of $\frac{freq_{i,d}}{Max_l(freq_{l,d})}$. Though 4 sampled documents might not be enough to represent the entire data set, they are helpful to give an intuition of the distribution of term scores in these documents. Clearly, the skewed distribution of term scores in both figures is useful for our solution: with a given threshold, the skewed distribution helps select fewer TTerms, thus reducing the forwarding cost.

With no real thresholds available for filters, we randomly generate thresholds through the uniform distribution and the exponential distribution with a given mean value. The purpose of using exponential distribution is to study how HiBloom can summarize relatively low thresholds. The generated values are used as personalized thresholds.

(3) Performance metric: We use two metrics to measure the performance:

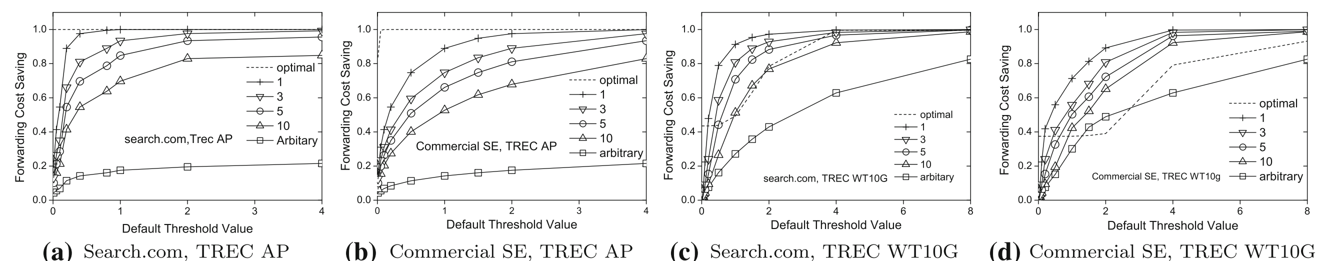
– *Forwarding cost saving ratio*: the saving ratio is 1.0 minus the ratio between the forwarding cost of STAIRS and the cost of the full-forwarding. The higher a saving ratio is, the less forwarding cost is used.
– *Document false dismissal ratio*: the ratio is 1.0 minus the ratio between the number of documents that subscribers actually receive by using STAIRS and that of documents that subscribers should receive. The higher a false dismissal ratio is, the more documents are missed.

9.2 Performance study of default forwarding

First, we conduct experiments to study the forwarding cost saved by default thresholds. In these experiments, we use TREC AP (and TREC WT10G) as documents, and each entry in the search.com query log (and the commercial SE query log) as a filter. During the document forwarding, we set up the number of terms in subscription filters as a system parameter $|f|_s$ equal to 10, 5, 3, and 1. By $|f|_s$, Algorithm 1 optimizes the selection algorithm by computing the sum of $|f|_s$ consecutive term scores. Instead, if setting $|f|_s$ with an arbitrary number, Algorithm 1 has to compute the sum of an arbitrary number of term scores (See Sect. 3.1). In addition, we compute the optimal forwarding cost saving ratios.

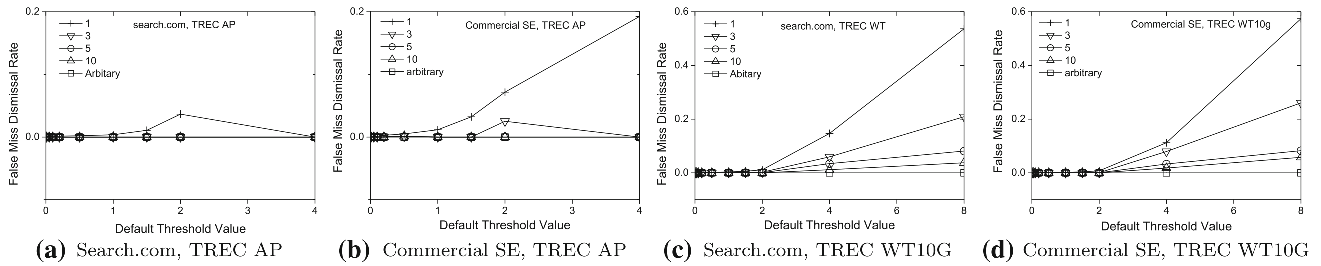
With the search.com query log as filters, Fig. 10a and c respectively plot the forwarding cost saving ratios for TREC AP and TREC WT10G document corpus; and Fig. 10b and d report the cost saving ratios with the commercial SE trace log as filters. Figure 10 gives the following four findings. (i) Algorithm 1, optimized for short filters, can save the forwarding cost. For example, in Fig. 10a for threshold 2.0, setting $|f|_s = 10$ can achieve 4.26 folds of the forwarding cost saving compared to the default selection algorithm with an arbitrary number of term scores; for threshold 0.1, setting $|f|_s = 1$ can achieve 7.96 folds of the forwarding cost saving. (ii) When the default threshold value in the x-axis grows larger, the forwarding cost saving ratio of y-axis also increases. This is because when a larger default threshold value is used, a smaller number of TTerms are selected by higher default thresholds. (iii) Since documents in TREC WT10G are relatively short articles composed by a smaller number of terms than TREC AP, the cost saving of TREC WG10G is lower than that of TREC AP. (iv) On the overall, both optimal cost savings and the cost savings by the default selection algorithm are increased with default thresholds, because given larger default thresholds, the default algorithm always selects a smaller number of terms with top scores.

Meanwhile, we measure false dismissal ratios. Using the search.com query log as filters, Fig. 11a and c respectively plot false dismissal ratios for TREC AP and TREC WT10g; with the commercial SE query log as filters, Fig. 11b and d plot false dismissal ratios for two aforementioned corpus.



(a) Search.com, TREC AP **(b)** Commercial SE, TREC AP **(c)** Search.com, TREC WT10G **(d)** Commercial SE, TREC WT10G

Fig. 10 Forwarding Cost Saving of Default Selection Algorithms: larger thresholds and smaller $|f|_s$ lead to more cost savings.



(a) Search.com, TREC AP **(b)** Commercial SE, TREC AP **(c)** Search.com, TREC WT10G **(d)** Commercial SE, TREC WT10G

Fig. 11 False dismissal ratio of Default Selection Algorithms: larger thresholds and smaller $|f|_s$ lead to more false dismissals.

From Fig. 11, we find the following two results. (i) Setting $|f|_s = 10$ or $|f|_s = 5$ produces a very low false dismissal ratio; even in Fig. 11c and d, when the default threshold value is unreasonably set with a large value 8.0, the false dismissal ratio is less than 0.0812. This result indicates that with larger threshold values, STAIRS can reduce the forwarding cost with a small false dismissal ratio as confirmed in Fig. 10. (ii) A relatively higher false dismissal ratio is produced when the number of short terms $|f|_s$ is smaller like 3, 2 or 1.

9.3 Performance study of adaptive forwarding

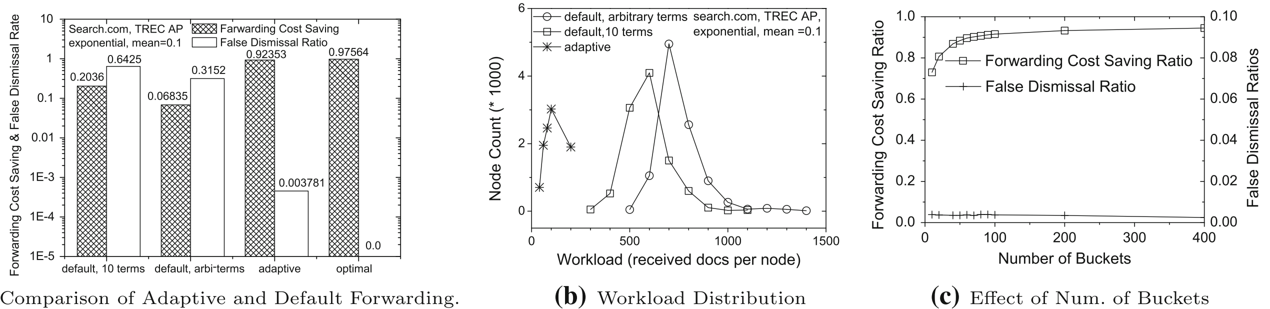
Considering that the results of various combinations of two query logs and two document corpus follow similar trends, in the rest of this section, we report experimental results by choosing one example combination (search.com as filters and TREC AP as documents). Also, we focus on the study of adaptive forwarding as it is the most general solution. In addition, the experimental results of default forwarding indicate that the forwarding cost heavily depends on the specified thresholds. If the threshold is lower, the forwarding cost is higher. Hence, we conduct experiments to study how the adaptive forwarding utilizes HiBloom to summarize filters.

9.3.1 Comparison of adaptive and default algorithms

In this experiment, thresholds are generated by the exponential distribution with a mean 0.1. To implement HiBloom, we set $k = 4$ hash functions. Note that to set a reasonable (or optimal) number of k for bloom filters, [13] gives an analytic study. Here, following Sect. 4.3, for the bloom filter inside HiBloom, we vary the number of bits based on the associated threshold. Next, we set $b = 100$ buckets by default (we will study the effect of the number of buckets in Fig. 12c). Finally, we set $|f|_s = 3$, according to the average number of terms per query in our query trace logs.

In Fig. 12a, the adaptive algorithm achieves the highest forwarding cost saving ratio 0.92353, and the lowest false dismissal ratio 0.00378 (except the optimal result). Instead, the default algorithm consumes the highest forwarding cost (i.e., least forwarding cost saving ratio) and the largest false dismissal ratio. From this experiment, we find that summarizing the filter information (i.e., query terms, filter thresholds and filter sizes) is critical to reduce the forwarding cost and false dismissal ratio. Besides, compared with the optimal result with the cost saving ratio and false dismissal ratio equal to 0.97564 and 0.0, respectively, the adaptive algorithm achieves only 5.34% less forwarding saving and 0.378% more false dismissals.

In addition, in this experiment, we compare the performance of HiBloom with biased assignment of bits (see Sect. 4.3) with HiBloom using standard bloom filters (i.e., the bloom filter of each bucket uses 2^{20} bits). Though both implementations achieve the same false dismissal ratio equal to 0.00378, the HiBloom using biased assignment of bits uses only 921 KB space cost; the HiBloom with standard bloom filters spends 11.2 MB data size. It validates the superiority



(a) Comparison of Adaptive and Default Forwarding. **(b)** Workload Distribution **(c)** Effect of Num. of Buckets

Fig. 12 Performance Study of Adaptive Algorithm, achieving largest cost savings, least false dismissals, and balanced workload

of HiBloom with biased assignment of bits over that with standard bloom filters.

9.3.2 Workload distribution

Figure 12b plots the workload distribution of the experiment above, where the x-axis represents the workload per node (measured by the number of received documents per node), inside the range of two neighbor values shown in the x-axis (e.g., from 100 to 200), and the y-axis represents the node count inside the load range of the x-axis. For comparison, we plot the load distribution of the default algorithm with arbitrary terms, default forwarding with $|f|_s = 10$, and the adaptive forwarding. For the load distribution, we are interested in the *overloading* issue: we consider that one node is overloaded when this node receives more than 2 times of the average load.

By the default algorithm for arbitrary terms, the average load is equal to 664 documents per node, and 1.58% nodes (i.e., 158 nodes) receives more than 1.200 documents. For $|f|_s = 10$, the average load is 581 documents per node, and only 0.35% nodes (35 nodes) receives more than 1.000 documents. By the adaptive algorithm, the average workload is 132 document per node, and no node receives more than 264 documents.

These results are explained as follows. In the document corpus, some terms frequently appear in documents, yet some terms rarely appear. Thus, frequent terms produce low *idf* values, and correspondingly small $s(t_i, d)$. By the default algorithm with arbitrary terms, a low selection ratio means that still some frequent terms are selected as TTerms (though the most frequent terms are excluded). Thus, a large number of documents (containing these frequent terms) are forwarded to home nodes of these frequent terms, suffering from the overloading problem. By the adaptive algorithm, more number of frequent terms are excluded, and their home nodes are free of the overloading issue. Obviously, the full-forwarding, which select all terms as TTerms, suffers from the most serious overloading issue.

9.3.3 Effect of HiBloom and maintenance cost

Figure 12c studies the effect of b, i.e., the number of buckets in HiBloom. When b grows from 10 to 400, the forwarding cost saving ratio is increased from 0.731 to 0.945. For $b = 60$, the saving ratio reaches 0.902; after that, the growth trend of the saving ratio is relatively smooth. Meanwhile, when b becomes further larger, the false dismissal ratio keep stable, around 0.00381. Note that a larger number of b leads to a higher cost saving ratio, but incurring a larger space cost. For example, when b grows from 40 to 400, the data size of HiBloom is linearly increased by around 10 times.

Finally, we measure the maintenance cost to propagate HiBloom across the DHT. We compare our approach with the one to broadcast the details of all filters (including query terms and filter thresholds) to every node in the DHT, where the broadcast message contains query terms, thresholds and unique filter IDs. For fair comparisons, both propagations are based on the scalable aggregation tree [46] (See Sect. 4.3), and we use the Zip compression technique to reduce the package size. In these settings, the overall cost of our approach is 285 KB/Msg · 10,000 Msgs = 2.72 GB; the overall cost of the broadcast approach is 2,187 KB/Msg · 10,000 Msgs = 20.87 GB. Considering that subscribers may tune thresholds and even change query terms, more maintenance cost is required by the broadcast approach. When the number of filters is very large, HiBloom, as a summary structure, consumes less space cost than broadcasting the details of all filters.

9.4 Performance study of the refinement algorithm

In this section, we study the performance of the refinement algorithm. Given the parameters listed in Table 3, we measure the number of selected TTerms, pruning ratio, maintenance cost, and the average hop count of forwarding documents. We implement the refinement structure with $m = 2^{12}$ bits and $k = 4$ hash functions for bloom filters in the refinement structure.

9.4.1 Effect of the number of documents

First, we study the effect of the number of documents, ranging from 10 to 1,000. Figure 13a plots the number of TTerms per document, selected by (i) the adaptive forwarding without the refinement algorithm (ii) the adaptive forwarding having the refinement, and (iii) the optimal number of TTerms. Since we use the same TTerm selection algorithm (i.e., Algorithm 2), the approaches with and without refinement obviously have the same number of TTerms. From this figure, we find that more documents first lead to more TTerms; however, the growth of TTerms becomes smooth, when the number of documents becomes larger. This trend is explored as follows.

Table 3 Parameters used in Sect. 9.4

Parameter	Range	Default
N: Node count in DHT	100–10,000	10,000
D: Num. of documents	100–1,050	4,00
F: Num. of filters	100–81,497	81,497
$\|f\|$: Filter size	1–10	3
k: # of hash functions in bloom filters		4
m: # of bits in refinement structure		2^{12}

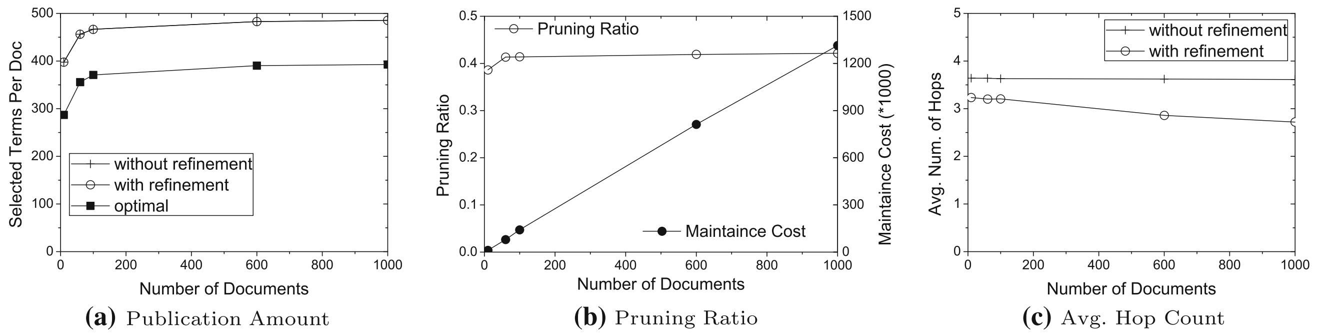


Fig. 13 Effect of the Num. of Documents: more documents lead to more TTerms, higher pruning ratio, and lower avg. hop count

When more documents are published, more document terms commonly appear in filters, leading to more selected TTerms. However, when the number of documents is large enough, the number of selected TTerms reaches its maximal value (i.e., the total number of query terms). It is because the number of filters does not change in this experiment. This trend is consistent with the optimal number of TTerms.

Figure 13b plots the average pruning ratio and maintenance cost (measured by the number of propagated refinement structures). The average pruning ratio remains 0.413. Meanwhile, due to the propagation of refinement structures, the maintenance cost increases with more documents. On average, 1312.47 refinement messages are propagated per document.

Figure 13c plots the average hop count. Because a document can be pruned before its arrival at a destination home node, the average hop count of the approach with refinement is smaller than that without refinement. For example, when the number of documents is 1,000, the average hop count of the approach with refinement is 2.719, yet for the approach without refinement, the average hop count is 3.613. Considering the trade-off between the paid maintenance cost and the gain of the reduced average hop count, we believe that the reduced average hop count is still beneficial to save the overall network traffic. The saved publication cost per document is 334 KB · 485,498/1,000 · (3.613−2.719)/1,024 = 141.57 MB, where 334 KB is the average size per publication, and 485,498 is the total number of the selected TTerms to publish 1,000 documents. Yet, the maintenance cost per document is only 1312.47 · 1.02 KB = 1.307 MB, where 1.02 KB is the average size of a refinement structure.

9.4.2 Effect of the number of filters

In this experiment, we study the effect of the number of registered filters. For the selected TTerms, Fig. 14a shows the results that are similar to Fig. 13a. However, when the number of filters is increased from 1,000 to 80,000, the ratio between the number of selected by the adaptive selection algorithm and the number by the optimal algorithm is also increased from 1.035 to 1.246. It means that more filters result in selecting more TTerms, thus incurring more forwarding cost. This is because more filters contain more query terms, and the corresponding HiBloom is less precise to select TTerms.

For the pruning ratio shown in Fig. 14b, more filters do not necessarily result in smaller pruning ratios. Actually, more selected TTerms (due to more filters, already shown in Fig. 14a) lead to propagating more refinement structures to intermediate nodes toward home nodes of the selected

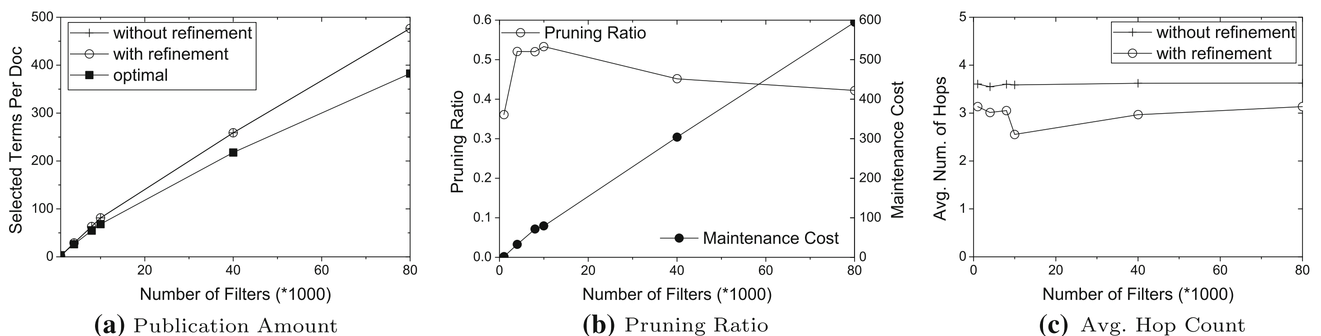


Fig. 14 Effect of the Num. of Filters: more filters lead to more TTerms, but unnecessarily less pruning ratio or higher avg. hop count

TTerms. Therefore, the pruning algorithm (i.e., Algorithm 4) prunes more documents, leading to a higher pruning ratio. Meanwhile, more filters incur less precision of the refinement structure and finally decrease the pruning ratio.

Due to the effect of the pruning ratio, the average hop count in Fig. 14c is first decreased to 2.554 at the point with 10,000 filters and increased smoothly to 3.135 at the point with 80,000 filters.

9.4.3 Effect of node count

Figure 15 shows the effect of the node count. Since the term selection algorithm (Algorithm 2) is independent upon the node count, there is no effect of the number of selected TTerms as shown in Fig. 15a.

Next, because the number of selected TTerms and the number of filters are not changed, the pruning ratio in Fig. 15b remains unchanged.

In Fig. 15c, for the approach without refinement, the average hop count is increased with the growth of the node count. Also, the average hop count of the approach with refinement is increased with the growth of the node count, because the pruning ratio keeps unchanged. For the maintenance cost, when the average hop count is increased, more refinement structures are correspondingly propagated, as shown in Fig. 15b.

9.4.4 Effect of filter size

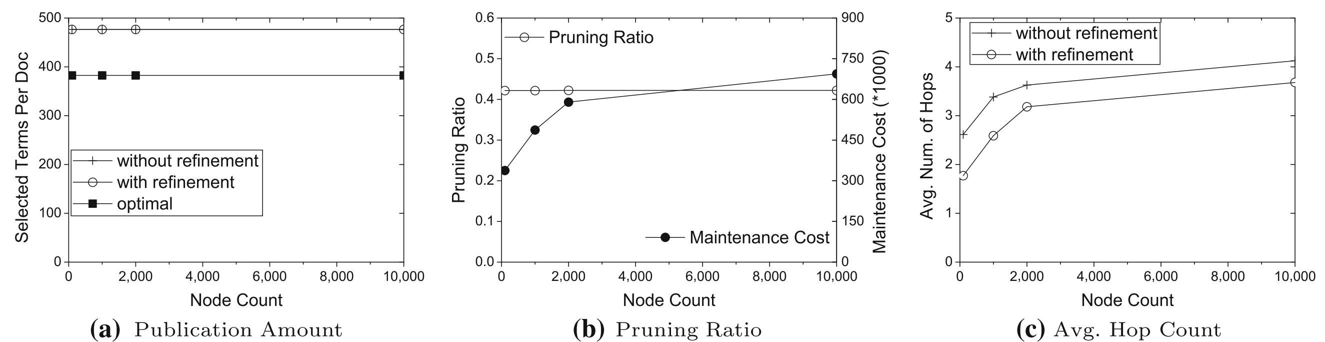
To study the effect of the filter size $|f|_s$, in Fig. 16, we change $|f|_s$ from 1 to 10. In particular, we are interested in the effect of $|f|_s$ in Algorithm 4. Therefore, during the term selection, we fix the filter size $|f|_s = 2$ of Algorithm 2 for fairness. As shown in Fig. 16a, the number of selected terms keeps stable for both approaches with and without refinement.

In Fig. 16b, the pruning ratio is decreased, and the maintenance cost is increased when the value of $|f|_s$ grows. For the false dismissal ratio, though without plotting in the figure, for $|f|_s = 1, 2, 5$ and 10, the false dismissal ratio is 0.163%, 0.0299%, 0.0281% and 0.0285%, respectively. This experiment indicates that $|f|_s = 2$ is enough for Algorithm 4 to achieve a high pruning ratio and low maintenance cost ratio, without sacrificing the false dismissal ratio.

In Fig. 16c, for the approach with refinement, due to less pruning ratio, the average number of routing hop count is increased from 2.65 to 3.28, when $|f|_s$ is changed from 1 to 10. Instead, the average hop count of the approach without refinement keeps stable, equal to around 3.63.

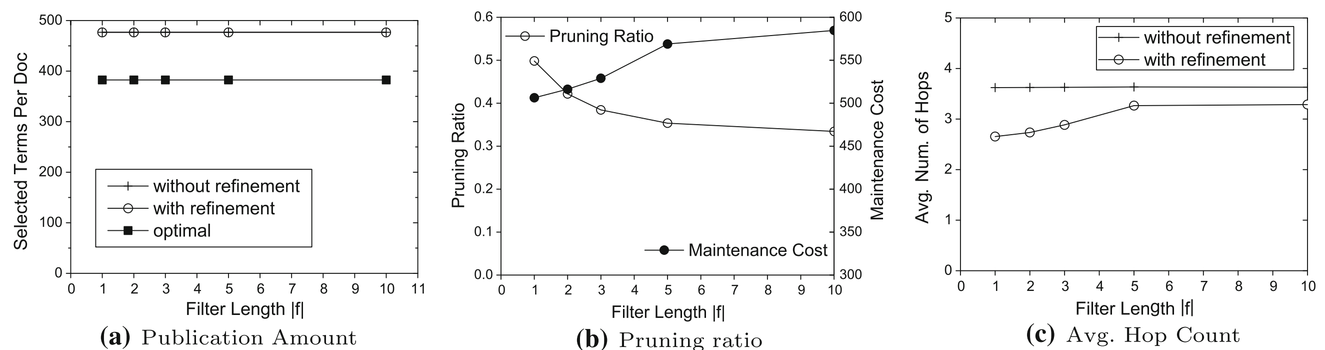
9.5 Performance study of the extended data model

During the implementation, we set $s = 10$, so that the rang of (0.0, 1.0] is divided by $2^s = 1024$ (even) intervals. The space size of a value-based bloom filter (VBF) with $s = 10$



(a) Publication Amount **(b)** Pruning Ratio **(c)** Avg. Hop Count

Fig. 15 Effect of Node Count: more nodes lead to more maintenance cost and higher avg. hop count



(a) Publication Amount **(b)** Pruning ratio **(c)** Avg. Hop Count

Fig. 16 Effect of Filter Size $|f|_s$: larger $|f|_s$ decreases the pruning ratio and increases the avg. hop count

is around 10 folds of the size of a standard bloom filter. For example, for $k = 4$ and the number of bits $m = 2^{16}$, the space size of a standard bloom filter is 8.08 KB, and the space cost of the VBF with $s = 10$ is 88 KB. To generate weight $w(t_i, f)$ of term $t_i \in f$, we randomly produce a numeric value within (0.0, 1.0]. The TREC AP is used as publication documents, and the search.com as subscription filters. We also use Table 3 to set up parameters (except for $D = 1,050$). Given these settings, we measure the performance of the extended selection algorithm (Algorithm 6) and extended pruning algorithm (given by Sect. 7.2).

For Algorithm 6, the selection ratio is 0.08633, and the forwarding cost saving ratio is 0.91377. Compared with the result of the simple data model (See Fig. 12a), more than 12.76% TTerms are selected. It is due to the fact that the extended selection algorithm utilizes the upper bounds of divided intervals as the exact weight values and selects more TTerms. Though with a higher selection ratio, a benefit is a smaller false dismissal ratio equal to 0.03182.

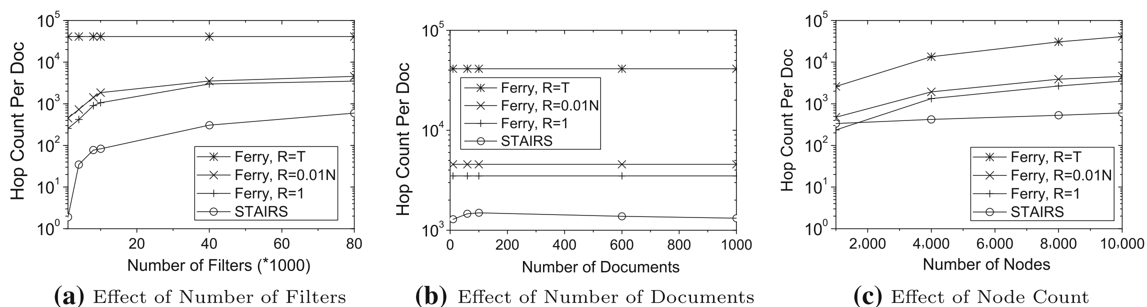
For the extended refinement scheme in Sect. 7.2, the pruning ratio is 0.368, the average hop count is 2.863, and the maintenance message number is 1558.21 per document. Since the refinement structure also utilizes VBF, the extended refinement scheme has smaller pruning ratio, larger hop count, and more maintenance messages than the simple data model. Clearly, if we increase s to a larger value, the extended refinement scheme in Sect. 7.2 can achieve a smaller selection ratio, and prune more irrelevant documents, but with the cost of a higher space cost.

9.6 Comparison with Ferry

In this section, as a complete solution to offer the full-text content filtering and dissemination, we compare STAIRS with Ferry [52]. For fairness, we implement the *PredRP* algorithm of Ferry on top of FreePastry. To ensure that Ferry can support the full-text document filtering and dissemination, we implement 3 variants of Ferry to set up R, i.e., the number of rendezvous nodes (RNs for short): (i) $R = 1$, i.e., only 1 RN, by considering that the schema s of Ferry contains only one attribute; (ii) $R = T$ RNs (where T is the total number of distinct query terms in all registered filters), by treating each distinct term as an attribute, and if T is larger than N (i.e., the total number of nodes in the DHT), then we reset $T = N$; (iii) $R = 0.01 \cdot N$ RNs. For any variant of Ferry with R RNs, T query terms are evenly distributed to R RNs, which on average manages T/R query terms per RN. Following the load balancing policy of Ferry, each node in the DHT is, on average, responsible for T/N query terms. In detail, if a rendezvous node r is responsible for term t_i, then all filters containing t_i are registered to r. If the number of extra filters is larger than T/N, those 1-hop nodes in the finger table of r (i.e., the direct neighbor set in FreePastry) will manage those filters. If r and all of those 1-hop nodes still cannot satisfy the load balancing requirement, then 2-hop nodes int the finger table of r, and even multiple-hop finger table nodes, cooperatively manage those filters. In addition, in all embedding trees (EMTrees for short), each non-leaf node utilizes HiBloom (i.e., equal to the summary filter in Ferry) to summarize all filters registered to the subtree rooted at this non-leaf node. Given these settings, following Ferry, filter f is registered to only one RN (i.e., actually to one EMTree), and document d is forwarded to all RNs. After arriving at a RN, d traverses the EMTree until all satisfied filters are found.

Still using Table 3 to set up parameters, based on the simple data model, we conduct experiments to compare STAIRS with Ferry and measure the overall hop count of forwarding a document to *all* satisfied filters (this metric covers both the publishing amount and unit-publishing cost). In Fig. 17a, we vary the number of filters from 1,000 to 80,000. When the number of filters grows, the hop counts of both STAIRS and Ferry are increased (except the case for Ferry with $R = N$). However, STAIRS consumes less hop count than Ferry variants. The reason is explored as follows.

– For Ferry with $R = 1$, only a single copy of document d is forwarded to the unique RN. However, the EMTree consists of $O(N)$ nodes. Though HiBloom can avoid traversing the whole EMTree, due to the high number of



(a) Effect of Number of Filters **(b)** Effect of Number of Documents **(c)** Effect of Node Count

Fig. 17 Comparison between STAIRS and Ferry: STAIRS typically consumes less hop count per document

terms per document (without using any selection algorithm as STAIRS), Ferry has to traverse the majority of branches in the EMTree, incurring 5.863 times of higher hop count than STAIRS;

- For Ferry with $R = N$, each EMTree consists of only one node (i.e., RN). Thus, no traversal of the EMTree is needed. However, Ferry has to forward each document to $R = T$ RNs (reset by $R = N$). Since each forwarding needs $O(\log N)$ hops, the overall hop count per document[2] is $N \log N$, and this is the largest cost among all approaches shown in Fig. 17a;
- For Ferry with $R = (0.01 \cdot N) = 100$, document d is forwarded to 100 RNs, and thus $(100 \cdot \log N)$ hops are consumed. When arriving at each RN, d still needs to traverse the associated EMTree. Thus, the overall hop count per document is the hop count used for forwarding the document to RNs, plus the hop count used for the traversals of embedding trees;
- Compared with Ferry, STAIRS can prune irrelevant documents *before* arriving at the home node of a TTerm. Instead, the EMTree traversal occurs *after* d arrives at a RN. Thus, STAIRS requires less hop count than Ferry does (for all 3 variants). In addition, in STAIRS, the home node of each TTerm is used as a RN to match incoming documents with locally registered filters. Due to the DHT hash function, the hashing ID of each TTerm is evenly mapped to home nodes across the whole DHT. Thus, there is no need[3] to form EMTrees to avoid unbalanced workloads. The benefit is no need to build an EMTree as required by Ferry, without traversing EMTrees.

Next, in Fig. 17b, we vary the number of documents from 10 to 1,000. For Ferry variants, no matter the number of documents, Ferry forwards each document to all RNs and then traverses EMTrees. Thus, the hop count per document is not affected by the number of published documents. Instead, in STAIRS, due to more available documents, the number of selected TTerms is increased (see Fig. 13a); meanwhile, the pruning ratio is decreased (See Fig. 13b). As a result, the hop count per document is first increase until the number of documents reaches 100; after that, such hop count is decreased. This experiment indicates that more number of published documents will finally helps STAIRS achieve a less forwarding cost.

Finally, in Fig. 17c, we vary the node count N from 1,000 to 10,000. As before, STAIRS consumes the least hop count among 4 approaches except $N = 1,000$, where Ferry with $R = 1$ consumes less hop count than STAIRS. This is because, for $N = 1000$, the traversal of EMTrees consumes at most $N = 1,000$ hop count per document; yet, for STAIRS, the overall hop count per document, related to the publication amount and unit-publishing hop count, is larger than a *small* N.

9.7 Summary and discussions

9.7.1 Summary of experimental results

(1) Patterns of filters and documents: First, the pattern of filters directly decides the performance of STAIRS. For example, if each filter contains a large number of query terms (instead of short filters), the experimental results in Sect. 9.2 indicate that using a larger filter size incurs a larger forwarding cost. Also, in Sect. 9.2, it is obvious that lower thresholds defined in filters lead to a higher cost due to more selected TTerms. In addition, Sect. 9.4 also shows that more filters also incur a higher forwarding cost.

In addition, for publication documents, Sect. 9.2 shows that STAIRS can achieve a high forwarding cost saving ratio for those articles composed by a large number of distinct terms (e.g., TREC AP); Sect. 9.4.1 shows that more publication documents help achieve less unit-publishing cost, due to more propagated refinement structures.

(2) P2P environments: We use the node count N to indicate the scalability of P2P networks. For example, Sect. 9.4.3 shows that a larger N leads to a higher unit-publishing cost, thus a higher overall forwarding cost; Sect. 9.6 further shows that a larger N leads to a larger overall hop count per document.

(3) Algorithms offered by STAIRS: Algorithms offered by STAIRS are important to the performance of STAIRS. Section 9.3 shows that the adaptive algorithm can help achieve the lowest forwarding cost, meanwhile with the lowest false dismissals. In addition, for the filter summary structure like HiBloom, Sect. 9.3 shows that a larger number of buckets obviously leads to a higher precision of HiBloom to summarize filters, but incurring a larger space cost. Similarly, in the refinement structure, more bits in bloom filters can also increase the precision, but with a higher space cost. Finally, setting smaller value of $|f|_s$ can reduce the forwarding cost, but incurring more false dismissals. Typically, using the average filter size (e.g., $|f|_s$=3) is enough to reduce both the forwarding cost and false dismissals.

9.7.2 Discussions

(1) Setting desirable filters: To assist subscribers with setting up favorable thresholds $\mathrm{T}(f)$, STAIRS can archive

[2] In this case, an improvement might be directly broadcast d to all nodes, incurring N hops, instead of $(N \log N)$ hops.

[3] The workload refers to the number of query terms per node. In terms of the number of filters per node as the workload, it could need to build an EMTree because the number of filters containing a query term could be uneven.

published documents and provide a test run facility based on the archived documents. By input query terms, subscribers run initial tests [47] with an initial threshold. Then, they interactively adjust thresholds to the desired level. When satisfied with the performance of this filtering, subscribers then set $\mathrm{T}(f)$ with the selected settings. This test facility helps subscribers further tune desirable thresholds even after the filter registration.

Besides setting desirable thresholds, a useful function to help subscribers specify desirable keywords is to recommend a subscriber with recently popular keywords that are used by other subscribers, keywords that appear in recently popular documents, and those keywords highly correlated with the current keywords specified in the defined filter.

Finally, to set up a default threshold, STAIRS can use the average value of all personalized thresholds. More complicated approaches consider the real values of $\mathrm{S}(f,d)$, the number of documents disseminated to subscribers, etc.

(2) Subscription privacy issue: Among all published documents, some of them are related to sensitive information (e.g., corporation or military) or political/religious affiliations. If a subscriber subscribes to these documents, it could indicate that the subscriber might be particularly interested in these documents. Besides providing the fine-grained content filtering and dissemination function, STAIRS is expected to protect the private subscription information. That is, subscribers expect that their private interest should not be exposed. Most existing works [23,36,21,35] on the privacy study of pub/sub systems focus on the access control, data confidentiality, and secure (and anonymous) routing (using cryptographic techniques), yet little has been conducted on the protection of subscription privacy. Our recent technical report [27] considers an attack model where some curious subscribers are compromised and colluded together with an untrusted broker. To defend against this attack, [27] proposes the k-subscription-anonymity model and optimally considers the trade-off between the privacy requirement and efficiency goal. Though [27] targets at a topic-based pub/sub system, the basic idea of [27] can be extended to the content-based pub/sub (it is our ongoing work) and can be further extended to STAIRS.

(3) Trade-off between usability and redundancy: A common issue of P2P networks is that a node can arbitrarily join or leave. As a result, filters and other information (e.g., the stored filter summary information, etc.), which are stored at such a node, will be lost. To handle this issue, a frequently used technique is to replicate stored objects (i.e., registered filters) to multiple nodes for redundancy. Among many proposed techniques [26,25,10,32,24], our previous work [26] derives an optimal solution which can trade-off the number of replicas and the average hop count. No matter what kinds of techniques are used, STAIRS, as an application over DHT-based P2P networks, can leverage them for better usability.

10 Conclusion and future work

In this paper, we propose a novel and simple content filtering and dissemination framework in DHT-based P2P networks, STAIRS. Different from previous works including distributed IF and the content-based pub/sub, STAIRS can effectively reduce the forwarding cost by the scheme of *full registration and partial forwarding*, and subscribers can receive the satisfying content with no duplicates.

For future work, we expect that STAIRS will be integrated with the existing RSS feed approaches [34] for realistic deployment and better usability. Also, we consider the top-k model, by which users subscribe to documents with the top-k highest values of $\mathrm{S}(f,d)$. Compared with the threshold model, the top-k model can avoid efforts to specify reasonable thresholds. Nevertheless, a benefit of the threshold model is that it can return quality-guaranteed documents.

Acknowledgment Funding for this work was provided by Hong Kong RGC NSFC Joint Project under project no. N_HKUST612/09 and NSFC Project Grants 60736013, 60873022, 60903053.

References

1. http://alert.live.com
2. http://en.wikipedia.org/wiki/special:statistics
3. http://www.google.com/alerts
4. Banavar, G., Chandra, T.D., Mukherjee, B., Nagarajarao, J., Strom, R.E., Sturman, D.C.: An efficient multicast protocol for content-based publish-subscribe systems. In: ICDCS, pp. 262–272 (1999)
5. Berry, M.W., Drmac, Z., Jessup, E.R.: Matrices, vector spaces, and information retrieval. SIAM Rev. **41**(2), 335–362 (1999)
6. Broder, A.Z., Mitzenmacher, M.: Survey: network applications of bloom filters: a survey. Int. Math. **1**(4), (2003)
7. Callan, J.P.: Document filtering with inference networks. In: SIGIR, pp. 262–269 (1996)
8. Cooper, B.F.: An optimal overlay topology for routing peer-to-peer searches. In: Middleware, (2005)
9. Cuenca-Acuna, F.M., Nguyen, T.D.: Text-based content search and retrieval in ad-hoc p2p communities. In: NETWORKING Workshops, pp. 220–234 (2002)
10. Dabek, F., Kaashoek, M.F., Karger, D.R., Morris, R., Stoica, I.: Wide-area cooperative storage with cfs. In: SOSP, (2001)
11. Deerwester, S.C., Dumais, S.T., Landauer, T.K., Furnas, G.W., Harshman, R.A.: Indexing by latent semantic analysis. JASIS **41**(6), 391–407 (1990)
12. Fabret, F., Jacobsen, H.-A., Llirbat, F., Pereira, J., Ross, K.A., Shasha, D.: Filtering algorithms and implementation for very fast publish/subscribe. In: SIGMOD Conference, pp. 115–126 (2001)
13. Fan, L., Cao, P., Almeida, J.M., Broder, A.Z.: Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans. Netw. **8**(3), 281–293 (2000)

14. Ganguly, S., Bhatnagar, S., Saxena, A., Izmailov, R., Banerjee, S.: A fast content-based data distribution infrastructure. In: INFOCOM (2006)
15. Ioannidis, Y.E.: The history of histograms (abridged). In: VLDB pp. 19–30 (2003)
16. Kukulenz, D., Ntoulas, A.: Answering bounded continuous search queries in the world wide web. In: WWW, pp. 551–560 (2007)
17. Li, J., Loo, B.T., Hellerstein, J.M., Kaashoek, M.F., Karger, D.R., Morris, R.: On the feasibility of peer-to-peer web indexing and search. In: IPTPS, pp. 207–215 (2003)
18. Lv, Q., Cao, P., Cohen, E., Li, K., Shenker, S.: Search and replication in unstructured peer-to-peer networks. In: SIGMETRICS (2002)
19. Michel, S., Triantafillou, P., Weikum, G.: Klee: A framework for distributed top-k query algorithms. In: VLDB, pp. 637–648 (2005)
20. Milo, T., Zur, T., Verbin, E.: Boosting topic-based publish-subscribe systems with dynamic clustering. In: SIGMOD Conference, pp. 749–760, (2007)
21. Nabeel, M., Shang, N., Bertino, E.: Privacy-preserving filtering and covering in content-based publish subscribe systems. Tech. Rep. (2009)
22. Nguyen, L.T., Yee, W.G., Frieder, O.: Adaptive distributed indexing for structured peer-to-peer networks. In: CIKM, pp. 1241–1250 (2008)
23. Opyrchal, L., Prakash, A., Agrawal, A.: Supporting privacy policies in a publish-subscribe substrate for pervasive environments. JNW (2007)
24. Ramasubramanian, V., Peterson, R., Sirer, E.G.: Corona: a high performance publish-subscribe system for the world wide web. In: NSDI (2006)
25. Rao, W., Chen, L., Fu, A.W.-C., Bu, Y.: Optimal proactive caching in peer-to-peer network: analysis and application. In: CIKM, pp. 663–672 (2007)
26. Rao, W., Chen, L., Fu, A.W.-C., Wang, G.: Optimal resource placement in structured peer-to-peer networks. IEEE Trans. Parallel Distrib. Syst. **21**(7), 1011–1026 (2010)
27. Rao, W., Chen, L., Yuan, M.: Towards efficient privacy-aware publish/subscribe. In: Hong Kong University of Science and Engineering, Department of Computer Science and Engineering, Technical Report, (2010)
28. Rao, W., Fu, A.W.-C., Chen, L., Chen, H.: Stairs: towards efficient full-text filtering and dissemination in a dht environment. In: ICDE (2009)
29. Ratnasamy, S., Francis, P., Handley, M., Karp, R.M., Shenker, S.: A scalable content-addressable network. In: SIGCOMM (2001)
30. Rose, I., Murty, R., Pietzuch, P.R., Ledlie, J., Roussopoulos, M., Welsh, M.: Cobra: Content-based filtering and aggregation of blogs and rss feeds. In: NSDI (2007)
31. Rowstron, A.I.T., Druschel, P.: Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Middleware (2001)
32. Rowstron, A.I.T., Druschel, P.: Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In: SOSP (2001)
33. Rowstron, A.I.T., Kermarrec, A.-M., Castro, M., Druschel, P.: Scribe: a large-scale and decentralised application-level multicast infrastructure. In: IEEE Journal on Selected Areas in Communication (JSAC), Vol. 20, p. 8 (2002)
34. Sandler, D., Mislove, A., Post, A., Druschel, P.: FeedTree: Sharing web micronews with peer-to-peer event notification. In: IPTPS, pp. 141–151 (2005)
35. Shang, N., Nabeel, M., Paci, F., Bertino, E.: A privacy-preserving approach to policy-based content dissemination. In: ICDE (2010)
36. Shikfa, A., Önen, M., Molva, R.: Privacy-preserving content-based publish/subscribe networks. In: SEC (2009)
37. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for internet applications. IEEE/ACM Trans. Netw. **11**(1), 17–32 (2003)
38. Stribling, J., Li, J., Councill, I.G., Kaashoek, M.F., Morris, R.: Overcite: a distributed, cooperative citeseer. In: NSDI (2006)
39. Tang, C., Dwarkadas, S.: Hybrid global-local indexing for efficient peer-to-peer information retrieval. In: NSDI, pp. 211–224 (2004)
40. Tang, C., Xu, Z.: pfilter: Global information filtering and dissemination using structured overlay networks. In: FTDCS, pp. 24–30 (2003)
41. Tang, C., Xu, Z., Dwarkadas, S.: Peer-to-peer information retrieval using self-organizing semantic overlay networks. In: SIGCOMM (2003)
42. Tang, C., Xu, Z., Mahalingam, M.: psearch: Information retrieval in structured overlays. In *HotNets-I*, (2002)
43. Terpstra, W.W., Kangasharju, J., Leng, C., Buchmann, A.P.: Bubblestorm: resilient, probabilistic, and exhaustive peer-to-peer search. In: SIGCOMM, pp. 49–60 (2007)
44. Tryfonopoulos, C., Idreos, S., Koubarakis, M.: Publish/subscribe functionality in IR environments using structured overlay networks. In: SIGIR, pp 322–329 (2005)
45. Xu, Q., Shen, H.T., Cui, B., Hou, X., Dai, Y.: A novel content distribution mechanism in dht networks. In: Networking, pp. 742–755 (2009)
46. Yalagandula, P., Dahlin, M.: A scalable distributed information management system. In: SIGCOMM, pp. 379–390 (2004)
47. Yan, T.W., Garcia-Molina, H.: The SIFT information dissemination system. ACM Trans. Database Syst. **24**(4), 529–565 (1999)
48. Yang, Y., Dunlap, R., Rexroad, M., Cooper, B.F.: Performance of full text search in structured and unstructured peer-to-peer systems. In: INFOCOM, (2006)
49. Zhao, B.Y., Kubiatowicz, J., Joseph, A.D.: Tapestry: a fault-tolerant wide-area application infrastructure, vol. 32, (2002)
50. Zhong, M., Shen, K.: Popularity biased random walks for peer-to-peer search under the square root principle. In: IPTPS (2006)
51. Zhu, Y., Hu, Y.: Efficient semantic search on dht overlays. J. Parallel Distrib. Comput. **67**(5), 604–616 (2007)
52. Zhu, Y., Hu, Y.: Ferry: a p2p-based architecture for content-based publish/subscribe services. IEEE Trans. Parallel Distrib. Syst. **18**(5), 672–685 (2007)