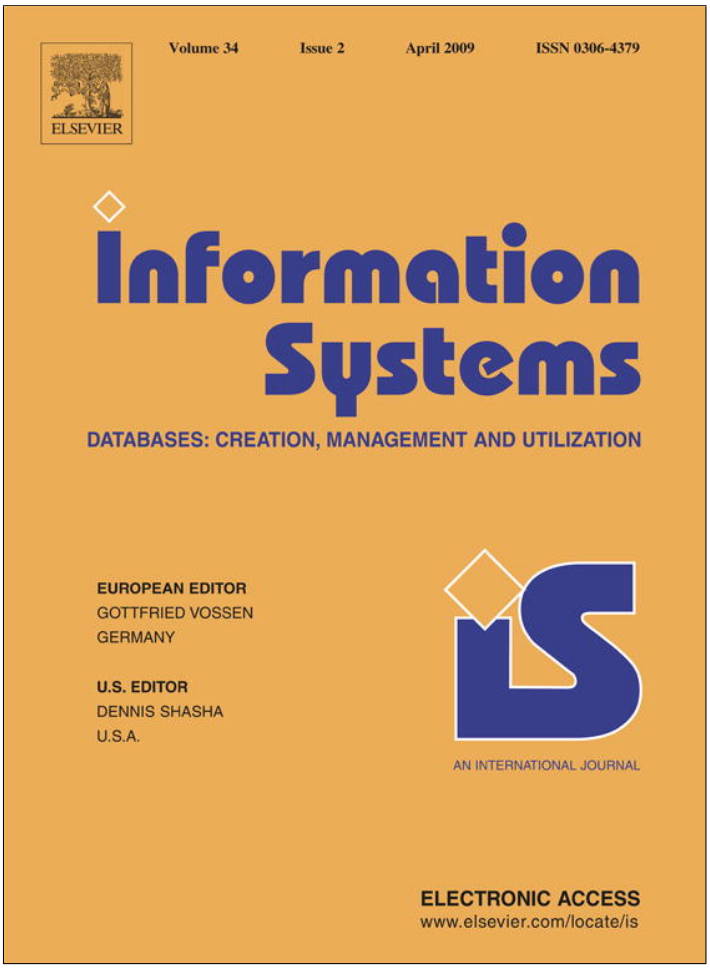

Information Systems 34 (2009) 290–303

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

PHIRST: A distributed architecture for P2P information retrieval ☆

Avi Rosenfeld [a,*,1], Claudia V. Goldman [c], Gal A Kaminka [b], Sarit Kraus [b]

[a] Department of Industrial Engineering, Jerusalem College of Technology, Jerusalem, Israel
[b] Department of Computer Science, Bar-Ilan University, Ramat-Gan, Israel
[c] Samsung Telecom Research Israel, Yakum, Israel

ARTICLE INFO

Article history:
Received 22 August 2007
Received in revised form
5 June 2008
Accepted 24 August 2008
Recommended by: P. Loucopoulos

Keywords:
Distributed databases
Information retrieval
Peer to peer systems

ABSTRACT

Recent progress in peer to peer (P2P) search algorithms has presented viable structured and unstructured approaches for full-text search. We posit that these existing approaches are each best suited for different types of queries. We present PHIRST, the first system to facilitate effective full-text search within P2P databases. PHIRST works by effectively leveraging between the relative strengths of these approaches. Similar to structured approaches, agents first publish terms within their stored documents. However, frequent terms are quickly identified and not exhaustively stored, resulting in a significant reduction in the system's storage requirements. During query lookup, agents use unstructured search to compensate for the lack of fully published terms. Additionally, they explicitly weigh between the costs involved in structured and unstructured approaches, allowing for a significant reduction in query costs. Finally, we address how node failures can be effectively addressed through storing multiple copies of selected data. We evaluated the effectiveness of our approach using both real-world and artificial queries. We found that in most situations our approach yields near perfect recall. We discuss the limitations of our system, as well as possible compensatory strategies.

1. Introduction

Full-text search, or the ability to locate documents based on terms found within documents, is arguably one of the most essential tasks in any distributed database [1]. Search engines such as Google [2] have demonstrated the effectiveness of centralized search. However, classic solutions also demonstrate the challenge of large-scale search. For example, a search on Google for the word, "a", currently returns over 15 billion pages [2]. Though Google's servers are capable of storing this magnitude of storage, this approach is infeasible for distributed solutions involving more limited devices.

In this paper, we address the challenge of implementing full-text search within peer-to-peer (P2P) network databases. Our motivation is to demonstrate the feasibility of implementing a P2P database comprised of resource limited machines, such as handheld devices. Thus, any solution must be keenly aware of the following constraints: *cost*—many networks, such as cellular networks, have costs associated with each message. One key goal of the system is to keep communication costs low. *Hardware limitations*—we assume each device is limited in its amount of storage. Any proposed solution must take this limitation into consideration. *Distributed*—any proposed solution must be distributed equitably. As we assume a network of agents with similar hardware composition, no one agent can be required to have storage or communication requirements grossly beyond that of other machines.

☆ This material is based upon work supported in part by ISF Grant # 1685/07.

* Corresponding author. Tel.: +972 8 926 8028.
E-mail addresses: rosenfa@jct.ac.il (A. Rosenfeld), c.goldman@samsung.com (C.V. Goldman), galk@cs.biu.ac.il (G.A. Kaminka), sarit@cs.biu.ac.il (S. Kraus).

[1] Portions of the work by the primary author was completed while at Bar-Ilan University.

0306-4379/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2008.08.002

Resilient—our assumption is that peers are able to connect and disconnect at will from the network. As a result, our system must be able to deal with peer failures, a concept typically referred to as churn [3,4].

To date, three basic approaches have been proposed for full-text search within P2P databases [5]. Structured approaches are based on the classic information retrieval (IR) theory [6], and use inverted lists to quickly find query terms. However, they rely on expensive publishing and query lookup stages. A second approach creates super-peers, or nodes that are able to locally interact with a large subset of agents. While this approach does significantly reduce publishing costs, it violates the distributed requirement in our system. Finally, unstructured approaches involve no publishing, but are unsuccessful in locating hard to find items [5].

In this paper we present PHIRST, a system for *P*eer-to-peer *H*ybrid *R*estricted *S*earch for *T*ext. This approach has three key contributions. First, PHIRST is the first system capable of performing distributed full-text search—something previously thought to be infeasible [1]. The key to PHIRST's success is its ability to restrict the amount of data needed to be published to execute full-text search. Not only does this ensure that the hardware limitations of agents' nodes are not exceeded, it also better distributes the system's storage. Furthermore, a peer's average data load actually decreases as peers with documents are added. Thus, the system becomes progressively more scalable as its size increases. Nonetheless, PHIRST is still able to effectively process full-text search through a hybrid approach that leverages the advantages of structured search (SS) and unstructured search (US) algorithms. PHIRST's limited published data are used to locate hard-to-find items. US is used to find common terms that were not published. Second, not only does PHIRST present a feasible approach for full-text search, but it also processes these searches with lower cost as well. We also present full-text query algorithms where nodes explicitly reason based on estimated search costs about which search approach to use, reducing query costs. Finally, we present how storing redundant copies of these entries can effectively deal with temporary node failures without the need of any centralized mechanism.

To validate the effectiveness of PHIRST, we used a real web corpus [7]. We found that the hybrid approach we present used significantly less storage to store all inverted lists than previous approaches where all terms were published [1,5]. Next, we used artificial and real queries to evaluate the system. The artificial queries demonstrated the strengths and limitations of our system. The unstructured component of PHIRST was extremely successful in finding frequent terms, and the structured component was equally successful in finding any pairs of terms where at least one term was not frequent. In both of these cases, the recall of our system was always 100%. The system's performance did have less than 100% recall when terms of 2 or more words of medium frequency were constructed. We present several compensatory strategies for addressing this limitation in the system. Finally, to evaluate the practical impact of this potential drawback, we studied real queries taken from IMDB's movie database [8] and found PHIRST was in fact effective in answering these queries.

2. Related work

Classical IR systems use a centralized server to store inverted lists of every term in every document within the system [6]. These lists are "inverted" in that the server stores lists of the location for each term, and not the term itself. Inverted lists can store other information, such as the term's location in the document, the number of occurrences for that term, etc. Search results are then returned by intersecting the inverted lists for all terms in the query. These results are then typically ranked using heuristics such as TF/IDF [9]. For example, if searching for the terms, "family movie", one would first lookup the inverted list of "family", intersect that file with that of "movie", and then order the results before sending them back to the user.

The goal of a P2P system is to provide results of equal quality without the need of a centralized server with the inverted lists. Potentially, the distributed solution may have advantages such as no single point of failure, lower maintenance costs, and more up-to-date data. Toward this goal a variety of distributed mechanisms have been proposed.

Structures such as distributed hash tables (DHTs) are one way to distribute the process of storing inverted lists. Many DHT frameworks have been presented, such as Bamboo [4], Chord [10], and Tapestry [11]. A DHT could then be used for IR in two stages: publishing and query lookups. As agents join the network, they need to update the system's inverted lists with their terms. This is done by every agent sending a "publish" message to the DHT with the unique terms it contains. In DHT systems, these messages are routed to the peer with the inverted list in $\log(N)$ hops, with N being the total number of agents in the network [4,10]. During query lookups, an agent must first identify which peer(s) store the inverted lists for the desired term(s). Again, this lookup can be done in $\log(N)$ hops [4,10]. Then, the agent must retrieve these lists and intersect them to find which peer(s) contain all of the terms.

Li et al. [1] present formidable challenges in implementing both the publishing and lookup phases of this approach in large distributed networks. Assuming a word exists in all documents, its inverted list will be of this length. Thus, the storage requirements for these inverted lists are likely to exceed the hardware abilities of agents in these systems as the number of documents grows. Furthermore, sending large lists will incur a large communication cost, even potentially exceeding the bandwidth limitation of the network. Because of these difficulties, they concluded that naive implementations of P2P full-text search are simply infeasible.

Several recent developments have been suggested to make a full-text distributed system viable. One suggestion is to process the SS starting with the node storing the term with the fewest peer entries in its inverted list. That node then forwards its list to the node with the next longest list,

where the terms are locally intersected before being forwarded. This approach can offer significant cost savings by insuring that no agent can send an inverted list longer than the one stored by the *least* common term [5]. Reynolds and Vahdat also suggest encoding inverted lists as Bloom filters to reduce their size [12]. These filters can also be cached to reduce the frequency these files must be sent. Finally, they suggest using incremental results, where only a partial set of results are returned allowing search operations to halt after finding a fixed number of results, making search costs proportional to the number of documents returned.

US protocols provide an alternative that are used within Gnutella and other P2P networks [13]. These protocols have no publishing requirements. To find a document, the querying node sends its query around the network until a predefined number of results have been found, or a predefined TTL (time to live) has been reached. Assuming the search terms are in fact popular, this approach will be successful after searching a fraction of the network. Various optimizations have again been suggested within this approach. It has been found that random walks are more effective than simply flooding the network with the query [14]. Furthermore, one can initiate multiple simultaneous "walks" to find items more quickly, or use state-keeping to prevent "walkers" from revisiting the same nodes [14]. Despite these optimizations, US has been found to be unsuccessful in finding rare terms [13].

In super-peer networks, certain agents store a disproportionate amount of inverted list data. Instead of all peers publishing inverted data over a distributed DHT network, agents send copies of their lists to their assigned super-peers. As agents are assumed to have direct communication with their super-peers, only one hop is needed to publish a message, instead of the $\log(N)$ paths within DHT systems. During query processing, an agent forwards its request to its super-peer, who then takes the intersection between the inverted lists of all super-peers. However, this approach requires that certain nodes have higher bandwidth and storage capabilities [5]—something we could not assume for our system.

Hybrid architectures involve using elements from multiple approaches. Loo et al. [15,16] propose a hybrid approach where a DHT is used within super-peers to locate infrequent files, and unstructured query flooding is used to find common files. This approach is most similar to ours in that we also use a DHT to find infrequent terms and US for frequent terms. However, several key differences exist. First, their approach was a hybrid approach between Gnutella ultrapeers (super-peers) and unstructured flooding. We present a hybrid approach that can generically use any form of structured or unstructured approaches, such as random walks instead of unstructured flooding or global DHTs instead of a super-peer system. Second, in determining if a file was common or not, they needed to rely on information locally available from super-peers, and used a variety of heuristics to attempt to extrapolate this information for the global network [15]. Since we build PHIRST based on a global DHT, we are able to identify rare-items based on complete information. Possibly most significantly, Loo et al. [16] only published the files' names, and not their content. As they considered full-text search to be infeasible for the reasons previously presented [1], their system was limited to performing searches based on the data's file name, and not on the text within that data. In contrast, the PHIRST system actually becomes more scalable as nodes are added and is thus the first system to facilitate effective full-text search even within large P2P databases. Finally, an earlier version of this work was previously published [17]. In addition to significant updates to the publishing and query algorithms presented in this paper, this version of PHIRST addresses issues of churn, something not addressed in our previous work.

3. PHIRST overview

First, we present an overview of the PHIRST system, how its publishing and query algorithms interconnect, and the variables used in them. While this section describes how information is published within the Chord DHT [10], PHIRST's publishing algorithm is generally presented in Section 4 so it may be used within other DHTs as well. Similarly, Section 5 presents query algorithms which select the best search algorithm based on the estimated cost of performing the search algorithms at the user's disposal. The selection algorithm (Algorithm 3) is generally written such that new search algorithms can be introduced without affecting the algorithm's structure. Only later, in Algorithm 4, we present how these costs are calculated specific to the DHT and US algorithms we used.

In order to facilitate structured full-text search for infrequent words, inverted file term data must be stored within structured network overlays such as Chord. Briefly, Chord uses consistent hash functions to create an m-bit identifier. These identifiers form a circle modulo 2^m. The node responsible for storing any given term is found by using a preselected hash function, such as SHA-1, to compute the hash value of that term. Chord then routes the term to the agent with the Chord identifier equal to or the successor (the next existent node) of that value [10]. For example, Fig. 1 depicts a simple example with three nodes, and an identifier space of 8 (2^3). Assuming the term hashes to a value of 6, it needs to be stored on the next node within the circular space, or on node 0. Assuming the term hashes to 1, it is stored on node 1.

The use of a consistent hash function within the Chord algorithm provides several positive qualities. First, it creates important performance guarantees, such as a $\log(N)$ average search length to find the node containing a certain term. Furthermore, nodes can be easily added (joins) or removed (disjoins) by inserting them into the circular space, and re-indexing only a fraction of the pointers within the system. Finally, the hash function used by Chord ensures that no agent will receive more than $O(\log(N))$ terms above the average amount stored by other nodes in the network [10], creating an equitable distribution of terms within the distributed system. We refer the reader to the Chord paper for further details [10].

In the PHIRST algorithms, we use the following variable names. Recall that N refers to the number of agents

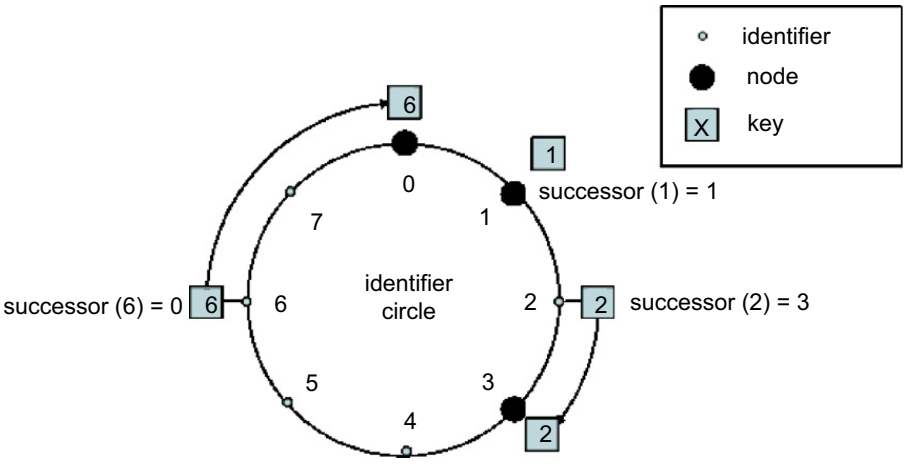

Fig. 1. An example of a Chord ring with $m = 3$. Figure based on Chord paper [10].

Table 1
Example of several words (terms within the DHT), and their inverted lists

$Term_i$	Address1	Address2	Address3	Address4	Address5	Address6	Address7
a	10.1.1.1	10.1.1.2	10.1.1.3	10.1.1.4	10.1.1.5	10.1.1.6	10.1.1.7
aardvark	10.13.132.45						
the	10.1.1.1	10.1.1.2	10.1.1.3	10.1.1.4	10.1.1.5	10.1.1.6	10.1.1.7
zoo	10.1.3.4	10.1.3.44	10.1.39.12				
zygote	10.7.12.45						

contained within the system. The PHIRST algorithms are based on knowledge of what this value for N is, and maintains a variable *NODE_COUNTER* to store this value. We refer to the total number of documents to be indexed for the full-text search as D. Unless otherwise noted, we assume that every node stores one document, and $N = D$. However, PHIRST's algorithms are not dependent on this simplification. In PHIRST's publishing phase, inverted file information must be added from a new document, Doc, for each of the terms $Term_i$ out of a total of *num_terms* terms, found on node ID_{SOURCE}. This information is sent to node ID_{DEST} to be published. ID_{DEST} can be found by using the successor function within Chord. Referring back to Fig. 1, assuming the term $Term_i$ hashes directly to the numeric value of one of the nodes (e.g. 0, 1, or 3), ID_{DEST} is assigned to this node. Otherwise, the next node (the successor) in the circular space becomes ID_{DEST}. ID_{DEST} is responsible for storing the inverted file information for this term, as well as any other term that was routed to it for storage. It keeps a counter, *TERM_COUNTER*, of the number of times the term $Term_i$ is found within the combined database of D documents.

For example, Table 1, provides an example of the inverted lists for five words stored on one node. Each row in the table represents a word, and the IP address(es) on the network where documents with that word can be found. Common words, such as "a" and "the" within the table, will produce much longer inverted lists than uncommon words such as "aardvark" and "zygote". Due to space restrictions this table only presents up to the first 7 inverted entries for each word, out of a potential number of D columns. Because word distribution within documents typically follow Zipf's law, some of the words within documents occur very frequently while many others occur rarely [18]. In an extreme example, one node may be responsible for storing extremely common words such as "the" and "a", while other nodes are assigned only rare terms.

While DHTs such as Chord are effective in equitably distributing terms over the system's nodes, they do not enforce equitable distribution of the amount of inverted file information per node. As such, Chord does balance the number of terms, $Term_i$, stored per node, but not the amount of data associated with these terms. Not only is this a major challenge in light of the distributed nature of the system, but it also prevents feasible full-text search [1]. The first key contribution of this paper is how we overcome these challenges by means of a publishing algorithm that limits the amount of inverted file information nodes must store, even for common words. We denote the size of the inverted file for term $Term_i$ as SIZE_OF_FILE($Term_i$), and limit this file's size to d entries. Details of PHIRST's publishing algorithms are found in Section 4.

However, PHIRST must still overcome the problem of how full-text queries can be properly performed without exhaustively indexing this information. We define the query task as finding a number of results, T, that match all query terms, or a total of *num_query_terms* terms, within the documents' text. Capping a query at T results is needed within US, as there is no global mechanism for knowing the total number of matches [5]. The second key

contribution of this paper is a novel querying algorithm that leverages between SS and US algorithms to effectively find matches despite the limit in the amount of data each peer stores. Furthermore, this algorithm selects between different types of search approaches based on estimated cost, reducing the cost of processing queries. Details of PHIRST's query algorithms are found in Section 5.

Finally, modifications to the above algorithms are necessary to address scheduled and unscheduled disconnects of nodes from the network. PHIRST incorporates a system for handling scheduled disconnects through an orderly mechanism for unpublishing inverted file information. It also contains replicated inverted file information for handling unscheduled node disconnects. Details of these algorithms are found in Section 6.

4. The publishing algorithms

First, once any agent joins the network, regardless of whether it has any documents to publish, it must update the variable *NODE_COUNTER*. We will see that this value is needed by the query algorithms described below. This can be done through an UPDATE_NODE_COUNTER function to send a request to the node responsible for storing this counter to update it by one. For simplicity, let us assume this global counter is stored on the first agent, ID_1, or $ID_{NODE\_COUNTER} = ID_1$. Thus, we assume all nodes perform the command, UPDATE_NODE_COUNTER(ID_1, 1) upon joining the network. We will explore variations of this assumption in Section 6 where we assume that multiple copies of this counter are necessary if node failures must be presumed and node ID_1 may not be available.

Next, every time an agent wishes to add a document to the network, it must publish the words in these documents as described in Algorithms 1 and 2. Note that there are two parts to this procedure. Algorithm 1 determines the terms in the document, Doc, that must be published, and where to send these terms in the DHT network. Algorithm 2, takes place on the receiving end, where node ID_{DEST} decides what information should be stored from node ID_{SOURCE}.

Algorithm 1. Publishing Algorithm (Document Doc)—Initiating Agent

1: *Terms* ⇐ Preprocessed words in Doc
2: *num_terms* ⇐ LENGTH(Terms)
3: **for** $i = Term_1$ to $Term_{num\_terms}$ **do**
4: ID_{DEST} ⇐ FindAddress(ID_{SOURCE})
5: ADD_TERM($Term_i$, ID_{SOURCE}, ID_{DEST})
6: **end for**

In Algorithm 1, node ID_{SOURCE} first generates a set of *Terms* it wishes to publish (line 1). Similar to other studies [5] we assume that the agent preprocesses its document to remove extraneous information such as HTML tags and duplicate instances of terms. Stemming, or reducing each word to its root form, is also done as it has been observed to improve the accuracy of the search [5]. Furthermore, as we detail in Section 7, stemming also further reduces the amount of information needed to be published and stored. In line 2 we initialize the *num_terms* variable to the number of unique terms to be published. The publishing agent, ID_{SOURCE}, then sends every term, $Term_i$, to be stored in an inverted list on peer ID_{DEST} (lines 3–6). This information will be used to create or update inverted files on the destination node. This node is identified via the function FindAddress in line 4 which can use Chord's previous described consistent hash function [10] or similar DHT implementation.

Algorithm 2. ADD_TERM ($Term_i$, ID_{SOURCE}, ID_{DEST})—**Receiving Agent**

1: **if** SIZE_OF_FILE($Term_i$) = 0 **then**
2: CREATE_FILE($Term_i$, ID_{SOURCE})
3: CREATE_TERM_COUNTER($Term_i$)
4: **else if** SIZE_OF_FILE($Term_i$) < d **then**
5: UPDATE_FILE($Term_i$,ID_{SOURCE})
6: UPDATE_TERM_COUNTER($Term_i$, 1)
7: **else**
8: UPDATE_TERM_COUNTER($Term_i$, 1)
9: **end if**

Next, the publishing algorithm consists of a second stage, depicted by Algorithm 2, where agent ID_{DEST}, responsible for storing inverted list information, must process these data. PHIRST enforces an equitable term distribution by requiring nodes to store a maximum of d entries in the inverted file for a given term. As lines 1–3 of the algorithm detail, assuming this is the first time this term has been found in all D documents, agent ID_{DEST} creates a new inverted file with this term (line 2) and a new term counter (line 3). Assuming this is an existing term with fewer than d entries, the location of ID_{SOURCE} is added to the existing inverted file (line 5) and the counter for $TERM\_COUNTER_i$ is incremented (line 6). Otherwise, d instances of that term exist and the location of the term is not added to the inverted file, but $TERM\_COUNTER_i$ is still incremented (line 8). This counter information is used by the query algorithms to determine the global frequency of this term.

Previous works require all term instances be published, and thus the length of the inverted files stored on nodes can grow unchecked. Because we limit each node so that it only stores a maximum d possible term instances, the storage requirements of the system are greatly reduced. Referring back to Table 1, let us assume the worst case, and all D documents in the system contain all terms and thus the variable *Terms* in Algorithm 1 will be the same for every Doc in D. Previous works require creating a maximum of D entries in the inverted lists represented in the table, while PHIRST requires only a maximum of d entries. Thus, in the worst case, PHIRST's storage requirement becomes $d * num\_terms$ instead of $D * num\_terms$. As we set $d \ll D$, we found these savings to be quite significant.

Theoretically, additional information about each term may be published, such as the position of the term or how many instances of the term exists within the document and this and similar information may be aggregated into a rating of the term to be published. This information may be especially important when more than d instances of

the term exist. The receiving agent, ID_{DEST}, could then decide which d term instances to store by continuously sorting scores of the terms it has, and maintaining only those with the highest d rating. In a similar vein, if more than d instances of $Term_i$ exist, it may be advantageous to store the most recent d documents, especially if turnover exists within nodes.

The performance guarantees of DHTs, such as Chord, ensure that the publishing algorithm runs at a fairly low cost. Because each node, ID_{SOURCE}, needs $\log(N)$ hops to find the agent, ID_{DEST}, responsible for storing the term's inverted list, the total number of messages needed to publish a document is of order $O(num\_terms * \log(N))$ where *num_terms* is the number of unique terms in that document. Note that the publishing algorithm described herein sends all terms, even those that in fact do not need publishing because they already contain d instances. For future work, we hope to study how nodes may be able to reduce this amount by knowing in advance which terms already have d terms.

5. The query algorithms

The query algorithms are called once any agent wishes to conduct a distributed full-text search. As Algorithm 3 describes, this process operates in two stages. First, the agent initiating the query retrieves the global frequencies of all search terms (line 2) and sorts them from least to most frequent (line 3). These values can be calculated by looking up the frequency of every term from the agent storing term $Term_i$. Referring back to Algorithm 2 note that the peer storing $Term_i$ has a counter with this value even if more than d instances of this term occurred. The frequency of every term is divided by the value *NODE_COUNTER* which can give an estimate as to how many nodes must be visited to find this term through an US. Finding these values requires one lookup of the value of *NODE_COUNTER* (which we previously assumed to be stored on agent ID_1), as well as a lookup of the frequencies of each term from the agent storing term $Term_i$. Referring back to Algorithm 2 (line 8) observe that the peer storing $Term_i$ has a counter with this value even if more than d instances of this term occurred.

Note that the expected frequency of terms is not necessarily equal to their actual frequency. For example, while the words "new" and "york" may be relatively rare, the frequency of "new york" is likely to be higher than the product of both individual terms. Thus, this naive approach may not be true for the actual combination of terms, a factor that may bias the algorithm towards using the wrong search approach, especially in borderline cases. This point is further discussed below regarding Algorithm 4.

Algorithm 3. Hybrid Query Algorithm (**T**, $Query_1 \ldots Query_{num\_query\_terms}$)

```
1:  space ⇐ ∞ {Used for initialization to all P2P nodes}
2:  Frequency_Array ⇐ Query_1 ... Query_num_query_terms
    {Frequency_Array is an unsorted array of query terms}
3:  Q_Array ⇐ Sorted Query Terms Least to most Frequent
    {Q_Array is the sorted array of query terms}
4:  for Query_i = Q_Array[1] to Q_Array[num_query_terms] do
5:    Frequency ⇐ Product(Frequency_Array[i] ...
      Frequency_Array[num_query_terms])
6:    Tradeoff ⇐ Calculate-Tradeoff(space, Query_i ...
      Query_num_query_terms, Frequency)
7:    if Tradeoff > 0 then
8:      Found ⇐ 0
9:      while (Found < T) AND (NOT Exhausted(space)) do
10:       Found ⇐ Found + Search-Unstruct(space, Query_i ...
          Query_num_query_terms)
11:     end while
12:     break
13:   else
14:     space ⇐ List(Query_i) ∩ space
15:     if i = Query_num_query_terms
16:       if LENGTH(space) > T then
17:         return first T list entries {Or NULL}
18:       else
19:         return all list entries
20:       end if
21:     end if
22:   end if
23: end for
```

Once the frequencies of all terms are known, the algorithm then reasons about which algorithm to select. This process iteratively calls the tradeoff function which we define below (Algorithm 4). If US is deemed less costly, all terms are immediately searched for simultaneously (lines 7–12). This type of search can either terminate because T matches have been found or the search space has been exhaustively searched. If SS is deemed less costly, that term's inverted list is requested, and the search space is intersected with the inverted list of the new term (the function List in line 14 returns the inverted list for the term). Assuming we have reached the last term (lines 15–21) we return the first T matches found after all terms have been successfully intersected. Once the SS identifies that fewer than T matches have been found (line 16) it returns all list entries (line 17). Note that line 17 also presents the option that failure or NULL is returned if less than T results were found.

This algorithm has several key features. First, the search process is begun starting with the least frequent term. This is done following previous approaches [5] to save on communication costs when using SS. Each successive peer receives the previously intersected list, and locally intersects this information with that of its term (line 14). As a result of this process the intersected lists become progressively smaller (or in the worst case remain the same length) and the maximum information any peer can send is bounded by SIZE_OF_FILE($Term_1$). Second, one might question why agents do not immediately return the entire inverted list of the terms they store, instead of first returning the term's frequency. Retrieving frequency information incurs a cost since we must find the frequency information for every term $Term_i$ out of a total of *num_query_terms* search query terms, and must also retrieve the value for the variable *NODE_COUNTER*. Thus, the cost overhead of finding this information is $num\_query\_terms * \log(N) + 1$ as the frequency of each term $Term_i$ can be found in the DHT in $\log(N)$, and we assume that *NODE_COUNTER* is found on ID_1 and consequently can be directly assessed in one hop. However, this

paper asserts that the information gained from this frequency information, such as bounding search costs to the size of the least frequent term, outweighs the search costs involved in processing the query in two stages. Finally, as the search goal is to return T results, the last node within a SS does not need to return its entire inverted list. Instead, it only needs to send the first T results. Accordingly, the maximal SS cost will be of order $(num\_query\_terms - 1)*\text{SIZE\_OF\_FILE}(Term_1) + T$.

Arguably the most important feature of this algorithm is its ability to switch between SS and US midway through processing the query terms. Even if SS is used for the first term(s), the algorithm iteratively calls the tradeoff algorithm (Algorithm 4) after each term. Once the algorithm notes that an US is cheaper, it immediately uses this search to find all remaining terms. For example, assume a multi-word query contains several common and uncommon words. The algorithm may first take the intersection of the inverted lists for all infrequent words to create a list f. The algorithm may then switch to an US within f to find the remaining common words.

Similarly, note that this approach lacks a TTL for its US that exist in other unstructured approaches [5]. We assume US is to be used only when the expected cost of using an US is low (see Algorithm 4). We expect this to occur when the US will terminate quickly, such as when: (i) the search terms are very common from the onset or (ii) an US is used to find the remaining common terms after a SS generated an inverted list of f terms.

We now address the search specific mechanism needed to identify which search algorithm will have the higher expected cost. This tradeoff depends on T, or the number of search terms desired, the costs specific for the different types of search options within the query algorithm, and d or the maximal number of inverted list entries published for each term. Algorithm 4 details this process as follows:

Algorithm 4. Calculate-Tradeoff (**T, space**, $Term_i \ldots Term_{num\_query\_terms}$, **Frequency**)

1: Expect-Visit $\Leftarrow$ T / Frequency {Number of nodes Unstructured search will likely visit}
2: **if** C_U* (Expect-Visit) $<C_S*$ (Sending(query-terms)) **then**
3: RETURN 1 {pure unstructured search}
4: **else if** SIZE_OF_FILE($Term_i$) $<$ d **then**
5: RETURN -1 {pure structured search for this term}
6: **else**
7: space $\Leftarrow$ List($Term_i$) $\cap$ space
8: RETURN 1 {Use unstructured afterwards}
9: **end if**

First, the algorithm calculates the expected cost of conducting an US. The expected number of documents that will be visited in an US before finding T results is: $T/Frequency$ (line 1), where Frequency is the product of frequency of all terms. We can compare this value to that of using a SS, whose cost is also known, and is proportional to the length of the inverted lists that need to be sent. We assume there is some cost, C_U associated with conducting an US on one peer. We also assume that some cost C_S is associated with sending one entry from the inverted list (line 2). Recall that the initial maximal SS cost is bounded by C_S* $((num\_query\_terms) - 1*$ $\text{SIZE\_OF\_FILE}(Term_1) + T)$ (see the previous description of Algorithm 3). The cost of the US can be calculated as $C_U * T/\text{Frequency}$. The ability to compare the expected cost of using both searches allows the algorithm to best decide how to proceed (lines 2–5).

For many cases, a clear choice exists with reference to which search algorithm would be best to use. Let us assume that $C_U = C_S = 1$, and that all documents have been indexed, or d is greater than or equal to the total number of documents in the system (D). When searching for common words, or Frequency is near 1, the cost of using the US is likely to be near T, as these words are likely to be found on the first several nodes searched. Processing the same query with SS will be more costly, as the inverted files' size is nearly equal to the number of documents and we assume T is much smaller than D. For example, assume $NODE\_COUNTER = 10\,000$, T is 10, each node stores 2 documents on average ($D = 10\,000 * 2$ or 20 000) and two terms are searched for with $\text{TERM\_COUNTER}(Term_1) = \text{TERM\_COUNTER}(Term_2) = 5000$ (Frequency $= 0.5 * 0.5$ or 0.25). Using US will find 10 results in approximately 40 node visits, while SS will require sending inverted files with 5000 terms. Note that in extreme examples, Frequency may be even much greater than 1, such as in the trivial case where only one node exists, but contains multiple documents with all terms. Conversely, for infrequent terms, say with one term occurring only 1 time, the cost of an US will be N or a number much larger than T, as all nodes must be searched before realizing T results could not be found. However, the SS will only cost a maximum of $num\_query\_terms - 1*\text{SIZE\_OF\_FILE}(Term_1) + T$, which is here bound by $\text{SIZE\_OF\_FILE}(Term_1)$ being only 1. Finally, SS is also the clear choice for queries involving one term. Note that in these cases, no inverted lists need to be sent ($num\_query\_terms - 1 = 0$), and only the first T terms are returned. The cost of using US will be greater than this amount (except for the exceptional case where the frequency of the term(s) is 1.0 or higher).

There are two reasons why the most challenging cases involve queries with terms of medium frequency. First, as previously mentioned at the beginning of Section 4, the expected frequency of terms is not necessarily equal to their actual frequency. As a result, the PHIRST approach is most likely to deviate from the optimal choice in these types of cases, especially when the values, C_U*(Expect-Visit) and C_S*(Sending(query-terms)) are similar. A second challenge results from the fact that we only publish up to d instances of a given term. In cases where inverted lists are published without limitation, e.g. d equals D, the second algorithm contains only two possible outcomes—either the expected cost is larger for a SS or it is not. However, our assumption is that hardware limitations prevent storing this large number of terms, and d must be set much lower than D. Consequently, situations will arise where we would *like* to use inverted lists, but as these files have incomplete indices, this approach will fail in finding results in position $d + \varepsilon$. While other options may be possible, in these cases our algorithm (in lines 6–8) takes the d terms from the inverted lists, and conducts an US for all remaining terms. In general, we found this approach to be effective as long as $T < d$, or the relationship, $T < d \ll D$

exists. We further explore the impact of this limitation in the results (7).

6. Dealing with network churn

The publishing and query algorithms address full-text search issues related to storage hardware limitations, methods for equitably distributing inverted lists, and search cost minimization. As we study real-world P2P databases, the system must additionally address temporary and permanent peer failures. This section provides a solution for this issue, known as churn, which is suitable for a distributed system.

6.1. The churn challenge

Churn can be defined as the turnover rate of nodes in the system over a given time period [3]. Based on previous work [3], we define churn (C) as follows: given a sequence of changes in the set of N peer nodes being available and unavailable, let U_i be the set of in-use nodes after the ith change, with U_0 as the initial set. Churn is the sum over each event of the *fraction of the system that has changed its state* in that event, normalized by run time t:

$$C = \frac{1}{t} \cdot \sum_{\text{events } i} \frac{|U_{i-1} \ominus U_i|}{N},$$

where $\ominus$ is the symmetric set difference.

Within many real-world networks, most churn events are likely to be caused by temporary changes of status where nodes are momentarily not in service, but will eventually return to operation. For example, a user might turn off her phone while sleeping, attending meetings, or going on vacations. We would expect these types of networks to have a rather high churn rate due to these types of events. A less frequent type of churn is likely when a node decides to permanently change its status, perhaps because a device breaks or when a user switches cellular carriers and receives a new number. While these events do not occur frequently, the node's information will be permanently lost if the device's data are not replicated.

Unstructured networks are not impacted by churn as long as the fluctuation of nodes' states still leaves the entire network with full connectivity [13]. As this method has no publishing or required routing of lookup information, it is unaffected by even very high churn rates. This has led some to claim that unstructured networks are most appropriate in these types of environments [13].

A variety of strategies have been proposed for dealing with churn in structured environments [3] to allow for continued connectivity once a node fails. One classic approach is to *reactively* fix the overlay network once a failure is detected. A second approach is to *periodically* check if nodes have failed. This can be done through constantly sampling neighbor nodes, and then preemptively replacing failed nodes before they cause a lookup failure. In environments with low churn levels the reactive methods perform better as they have no inherent communication overhead. However, in environments with even moderate churn levels, periodic methods perform significantly better [4].

It is important to differentiate between maintaining the connectivity of live nodes within the base DHT network, and the ability to find the data once stored in the network. Periodic approaches are better at handling churn, or finding nodes that are still participating in the network. However, our system must additionally maintain lost nodes' published inverted list information, something DHTs typically cannot do (refer back to the end of Section 3). We now present a solution for this challenge.

6.2. Addressing churn in a P2P application

First, we present Algorithms 5 and 6 to deal with planned types of disconnects, or when node ID_{SOURCE} can orderly remove the inverted list information it has published. While we recognize that planned disconnects will not represent all churn events within a real system, this algorithm utilizes key elements of PHIRST. We then generalize this approach for dealing with unplanned churn events.

Algorithm 5. Unpublishing Algorithm (Document Doc)–Initiating Agent

```
1:  Terms ⇐ Preprocessed words in Doc
2:  for i = Term_1 to Term_num_terms do
3:    ID_DEST ⇐ FindAddress(Term_i)
4:    REMOVE_TERM(Term_i, ID_SOURCE, ID_DEST)
5:  end for
6:  SUCCESSOR ⇐ FIND_SUCCESSOR(ID_SOURCE)
7:  for j = File_1 to File_NUM_FILES do
8:    COPY(File_j, SUCCESSOR)
9:  end for
```

Algorithms 5 and 6 address the actions a peer ID_{SOURCE} must perform before a planned disconnect (in Algorithm 5), and those ID_{DEST} must perform in removing ID_{SOURCE}'s terms (Algorithm 6). Note the strong similarity between these unpublishing algorithms, and the previously described publishing algorithms (Algorithms 1 and 2). In both cases, Similarly, the function REMOVE_TERM in line 4 of Algorithm 5 and UPDATE_TERM_COUNTER in line 4 of Algorithm 6 closely parallel the ADD_TERM function in line 5 of Algorithm 1 and the UPDATE_TERM_COUNTER functions in lines 6 and 8 of Algorithm 2. The purpose of the function REMOVE_TERM is to remove the entry of the disconnecting peer, ID_{SOURCE}, from the inverted list stored on peer ID_{DEST}. Note that these unpublishing algorithms also contain several new types of actions. In lines 6–9 of Algorithm 5, ID_{SOURCE} must copy the inverted lists it currently stores (out of a total of NUM_FILES inverted lists) to a new node. To do this, first we find the successor node within the overlay network (line 6) of Algorithm 5. DHT networks such as Chord [10] provide the implementation for the FIND_SUCCESSOR function referred to. Then, every one of the inverted files currently stored on ID_{SOURCE} (which number NUM_-FILES) are transferred over to the successor node. From this point onward, this node will have the responsibility of responding to queries for any of the terms contained in these inverted files. In Algorithm 6, peer ID_{DEST} removes

$Term_i$ if it had stored this value in its inverted file for this term (within the maximum of d terms it had stored). This check is done through the EXISTS function in line 1. Finally, after the peer ID_{SOURCE} has performed Algorithm 5 for all documents it had stored, or if a disconnecting node never had stored any documents, it must update the variable NODE_COUNTER of the global number of documents. However, here this value must be reduced through performing the function, UPDATE_NODE_COUNTER($ID_{NODE\_COUNTER}, -1$) instead of the function call UPDATE_NODE_COUNTER($ID_{NODE\_COUNTER}, 1$) used upon joining the network.

Algorithm 6. REMOVE_TERM($Term_i$, ID_{SOURCE}, ID_{DEST}) —**Receiving Agent**

```
1:    if EXISTS(Term_i, ID_SOURCE) then
2:      UNSTORE(Term_i, ID_SOURCE)
3:    end if
4:    UPDATE_TERM_COUNTER(Term_i, -1)
```

However, our assumption is that most churn effects result from temporary and unplanned failures where the failing node will not issue this unpublish command. In order to address this point, we present a solution where inverted list data are replicated to handle failures.

While the DHTs pointers need to be immediately updated in case of a node failure, however, temporary, that node's data do not need to be copied if a set of backup copies exist. PHIRST relies on this set of backups with the assumption that the node's failure was temporary, and it will soon function again in the network. This saves communication costs in copying inverted list data, assuming that node does soon rejoin the network.

These data replicas can be easily created during the publishing stage. We modify the publishing algorithm (Algorithm 1) to send its data to k peers instead of just one. We refer to the k copies of a group of R redundant nodes, $R_1, \ldots, R_k$, where all nodes receive the same publishing data. Note that DHTs such as Chord [10] can enable this functionality by sending a publishing message to the normal hashed peer ID_{DEST} within the Chord ring, and the next $k-1$ peers as well or, ID_{DEST}, $ID_{DEST}+1, \ldots, ID_{DEST}+k-1$. This is one approach to allow nodes to easily find the k copies. Consequently, line 5 of Algorithm 1 which stated: ADD_TERM($Term_i$, ID_{SOURCE}, ID_{DEST}) should be modified to become lines 5–7 of Algorithm 7.

Algorithm 7. Modification Publishing Algorithm for Churn (Document Doc)—Initiating Agent

```
1:    Terms ⇐ Preprocessed words in Doc
2:    num_terms ⇐ LENGTH(Terms)
3:    for i = Term_1 to Term_num_terms do
4:      ID_DEST ⇐ FindAddress(ID_SOURCE)
5:      for a = 0 to k do
6:        ADD_TERM(Term_i,ID_SOURCE, ID_DEST + a)
7:      end for
8:    end for
```

For example, assume $k=2$, or two copies of each inverted list are stored. Referring back to Fig. 1, all inverted lists that would normally be stored only on node 0, are now stored on nodes 0 and 1, inverted lists for 1 are stored on 1,3, etc. A temporary failure of node 1 will be handled by its successor, namely node 3, which also stored the same inverted list data. Join actions remain similar to those previously presented for DHTs with the exception that here a joining node must also be updated with the data currently being stored on the redundant nodes $R_1, \ldots, R_k$ that it is now joining. After these data are stored, the last node of the redundant set (R_k) can erase these data.

In a dynamic system, this approach will need to address two additional issues. First, care must be taken to address churn changes within the group of k redundant nodes after they are formed. For example, assume a new document is published, but node R_1 is temporarily unavailable. The data should still be published on the remaining $k-1$ nodes which now have the most updated version of the inverted lists. Once node R_1 becomes available again, it must receive the updated information. Second, we assume that most nodes become unavailable only temporarily, and thus a failure of a node, H, should not be a reason to immediately find a new node to replicate the data. However, after a certain time period, M, it must be assumed that node H has in fact left the system, and a new node must be found to store k nodes.

Algorithm 8. Replicating Data Under Churn (Node H)

```
1:    Start ⇐ Randomize start time
2:    for Time = Start to M step L do
3:      if Up(H) then
4:        Lock(H)
5:        Update(H)
6:        return
7:      end if
8:    end for
9:    Replace(H) {Assume Node C has left the network}
10:   UPDATE_NODE_COUNTER(ID_NODE_COUNTER, −1)
```

Algorithm 8 outlines these two steps. The precondition for this algorithm is that a group of k nodes has already published the application data (inverted lists) as per Algorithm 7, and these nodes are aware of the others' existence. Within Chord this can be done through predecessor and successor pointers. Also, this algorithm is called once the redundant node set notices that a member of the set is not working (node H). We assume this is done by periodically sampling, which has previously been shown to be effective [4]. Next, we assume the remaining nodes mark what data have been published after node H failed, and can thus update node H once it becomes available.

Based on these preconditions, the algorithm operates as follows. Once a failure has been detected, the remaining nodes monitor the down node for a total time period of M (line 2 of the algorithm). Each node checks if the failed node resumes operation every L time units. Note that each node randomizes its start time (line 1). Consequently two or more nodes do not unnecessarily monitor node H during the same time period. Assuming a node has found that H has resumed functioning, it updates node H with the information it missed and the algorithm terminates (lines 3–7). Immediately before performing this Update process, this node places a lock on H (line 4), so no other

node can update it simultaneously. This type of concurrency control mechanism is often used within databases [19]. Note that as part of the Update process, this node should also notify the other nodes in the replication set, $(R_1,\ldots,R_k)$, that there is no need to update node H with this information as well. Finally, if the algorithm reaches line 9, we assume node H is permanently down. As Chord operates by searching successor nodes for information, this replacement node must be taken from the next node not currently in the set $(R_1,\ldots,R_k)$, or node R_{k+1}. Additionally, we must reduce the variable *NODE_COUNTER* which stores the number of nodes in the system. This is done in line 9 of the algorithm.

However, comparing this approach to Algorithms 5 and 6 reveals two challenges. In Algorithm 5 a disconnecting node is assumed to be able to initiate an orderly disconnect, enabling a proper removal of all of its entries from the inverted lists. However, as the failures in this case are unpredictable, this is not possible with Algorithm 8. Furthermore, in line 10 of Algorithm 8 and throughout the paper we refer to $ID_{NODE\_COUNTER}$ as the peer responsible for storing information about the total number of nodes in the system. For simplicity, we had assumed this counter is stored on the first agent, or ID_1 (see the beginning of Section 4). However, this assumption must be changed to account for possible churn effects on this peer. One solution is to replicate this value k times to deal with churn, and refer to $ID_{NODE\_COUNTER}$ as this set of peers.

The value for k is a tunable parameter that must be set with care. As we deal with hardware with limited storage and assume communication is costly, care must be taken to refrain from sending unnecessary data. However, sufficient data replicas must be present to make the probability that all k peers will simultaneously fail extremely small.

Fortunately, the average system churn is typically a known or measurable quantity, and can be used to set the value of k. Assuming an average churn rate of p exists within the system, the probability the query algorithm will not find an inverted list given k redundant copies is p^k. For example, assume an average churn rate of 0.5, or 50% of the nodes will be unavailable in any given time period. The probability all five nodes will be down simultaneously is 0.5^5, or 0.03, and the probability at least one will still function is 1 − Probability(Failure_All), or 0.97. As the next section details, computing these probabilities are an effective guideline for setting k.

7. Experimental results

In this section we present the experimental results used to validate the effectiveness of the algorithms in this paper. As our research goal was to check if PHIRST is appropriate for medium sized newsgroups, we chose a corpus of 2000 real movie websites to conduct our experiments [7]. The results from the publishing experiments demonstrate that PHIRST actually becomes more feasible as more documents and agents are added to the network. We also created two types of query experiments. In one group we created artificial queries based on the frequencies of words. This experiment demonstrated the theoretical strengths and weaknesses of PHIRST. We also studied real movie queries based on the Internet Movie Database [8]. These experiments demonstrated that any weakness in PHIRST is likely to be insignificant in handling real queries.

7.1. Publishing experiments

Recall that the publishing algorithm is based on storing a maximum of d entries in a given term's inverted list. We simulated the publishing process to study how this parameter affected the average number of stored inverted entries with and without term stemming. Fig. 2 displays the average number of inverted terms (Y-axis) in groups of 50, 250, 500, 1000 and 2000 agents (X-axis). We assumed that every agent published 1 document taken from the movie corpus [7]. In the top graph, we used the Paice stemming algorithm [20] on each term before storing it. The bottom graph published each term without stemming. In both graphs we also ran the publishing algorithm with $d = 25$ and 75.

Several interesting results can be seen from this graph. First, on average stemming saved approximately 50 words per document. This is because stemming lumps similar words, reducing the number of unique words occurring per document. Second, note the publishing algorithm has progressively larger storage savings as the number of nodes grows. Assuming $d = D$, all terms will be stored, and no publishing gain will be realized by using the PHIRST approach. However, assuming d is kept fixed, the more documents that are added, the gap between d and D grows. This results in progressively more words exceeding the d threshold, and additional entries of these words no longer need to be stored. As a result, the publishing algorithm becomes *more* scalable the more nodes that are added, making full-text search feasible even in very large P2P databases.

Finally, in this experiment we assumed each node had 1 document to publish. We also ran this approach with more dense (e.g. 2 documents per node) or more sparse (e.g. 1 document every 2 nodes) network assumptions. As one would expect, the number of terms each node stores is proportional to the total number of nodes. For example, Table 2 shows the sparse assumption of 1 document published for every two nodes. These values are identical to those in Fig. 2 multiplied by a factor of 0.5.

We also found a Zipfian distribution of terms with a long tail of infrequent terms (see Fig. 3). Similar distributions have been found in P2P systems for items such as file frequency [15,16] and term frequency [18]. The storage saving results we found were from words with frequencies greater than d, or the terms towards the head of this distribution.

7.2. Query experiments

We first conducted query experiments based on artificial queries chosen according to term frequency. Fig. 3 displays the rank order of all words within the 2000 document corpus (a total of approximately 22 000 words)

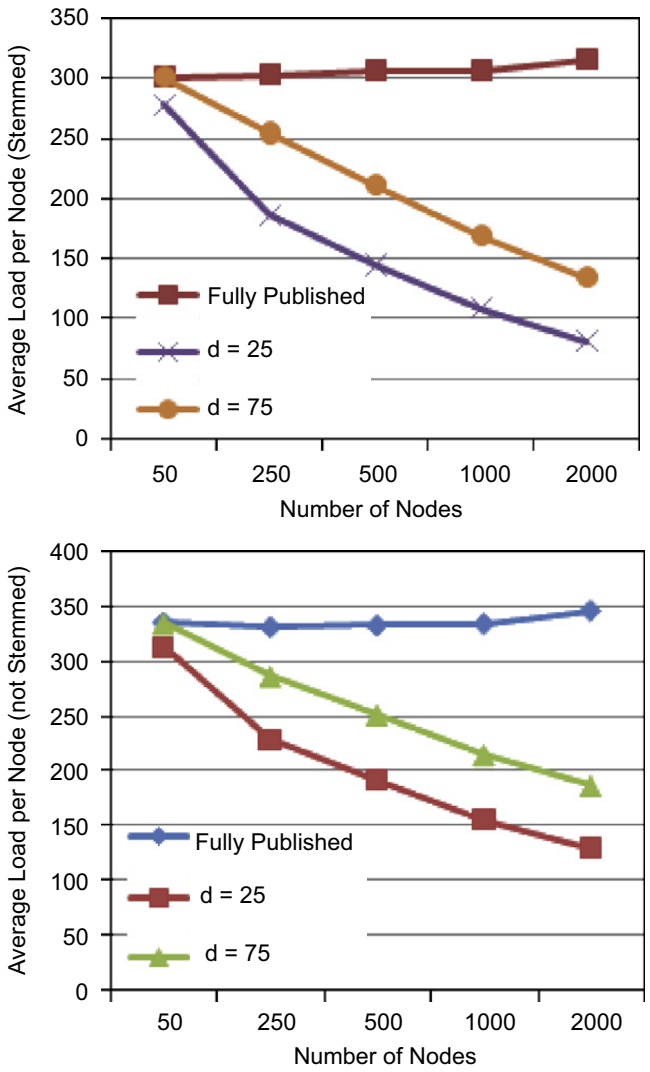

Fig. 2. A comparison of the publishing requirements of full publishing versus publishing limited to $d = 75$.

Table 2
Average number of inverted list entries if one document was published for every two peers

Number of nodes	50	250	500	1000	2000
Fully published	150.43	151.51	153.13	153.13	157.83
$d = 25$	138.84	93.11	72.17	53.97	40.605
$d = 75$	150.43	127.14	105.72	84.38	67.035

based on the words' frequencies. We considered words of high frequency if they appeared in 30% or more of the documents. There were 200 words in this category. Note that high frequency words are not just "stop" words like "the", "and", or "a", but can be specific to the corpus. For example, these words included movie specific terms such as "character", "play", and "plot". At the other extreme, we defined low frequency words as those appearing 50 times or less (frequency 2.5% or less). The large majority of terms were within this category due to the long tail of the term distribution. Finally, we assumed medium frequency words were those between the above extremes.

We created paired terms (2 terms) of all permutations of these categories. This involved words, both with high frequency (HH), both with low frequency (LL), both with medium frequency (MM), low high combinations (LH), low medium combinations (LM), and medium high combinations (MH). Note that the order of the words does not have an impact on the query algorithms since the terms are first sorted by these algorithms based on their frequency. For example, the low medium category (LM) is consequently equivalent to the medium low one (ML).

Next, we generated 1000 artificial queries from each category. We studied how many results were returned from each of four search algorithms. The SS algorithm published all terms and sent these indices between agents as needed during queries. The US algorithm used no publishing and used a random walk approach to find query results. In the used implementation, a random node was selected to begin the random walk, and assumed a fully connected graph allowing free passage between nodes. The TTL = 100 algorithm used the same US, but terminated after visiting 100 agents. Finally, the hybrid PHIRST approach implemented the publishing and query algorithms described in this paper. In these experiments we used a value of $d = 75$ in the PHIRST method.

Table 3 displays the average number of nodes visited (in the case of US) and/or the inverted list entries sent (for SS) for finding 20 matches from each query ($T = 20$). For simplicity, we assumed that the costs of visiting nodes through US, and sending inverted list entries are equal, or $C_U = C_S$. As expected, we found that the SS is the most expensive method for finding common terms; where the US is the most effective. Conversely, SS is the most effective in finding rare terms. As one might expect, the hybrid PHIRST approach operates similarly to SS in finding rare terms (LL) and US in finding common items (HH). This indicates the success of this approach in selecting the best search algorithm. Note that in middle categories (for example MH) this approach incurred significantly lower costs than the SS and US algorithms it is based upon. PHIRST saves costs by only sending a maximum of d entries even when SS is deemed necessary. Furthermore, this approach switches between the SS and US methods as needed, saving additional costs. Note that these results do not include the costs associated with looking up terms' frequency information. Recall from Section 5 that these costs are bounded by $num\_query\_terms * \log(N) + 1$ where num_query_terms is the number of query terms (in this case 2) and N is 2000 ($\log(N)$ is approximately 11). However, actual Chord implementations have found that the actual cost is often much lower and is dependant on the actual implementation of the DHT network [5]. Consequently, we do not include this cost in the results.

We also studied the impact of the number of documents per node (document density) on these costs. The US is most affected by the density of the documents. For example, assuming each agent stores two documents, the cost of using this search algorithm will be half. Conversely, sparse networks make US less appealing. This tradeoff does come to light within the unstructured

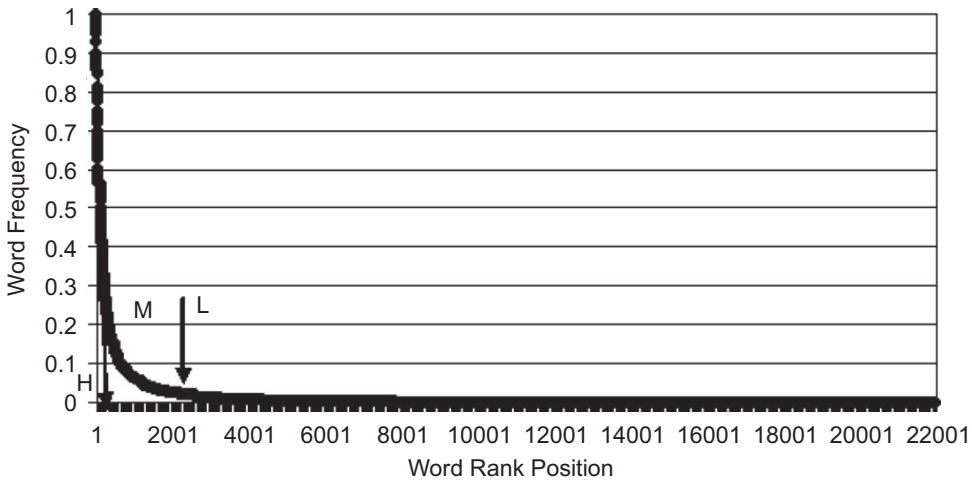

Fig. 3. Distribution of terms by rank order within movie corpus documents.

Table 3
A comparison of the cost levels of SS, US, TTL, and PHIRST methods in LL, LM, LH, MM, MH, and HH artificial queries

	SS	US	TTL = 100	PHIRST
LL	1466	2 000 000	100 000	1466
LM	2206	2 000 000	100 000	2142
LH	3177	1 987 754	100 000	2010
MM	20 732	1 865 474	99 953	13 256
MH	60 188	234 211	95 624	18 075
HH	871 986	19 746	20 077	19 995

Results for the case where $C_U = C_S = 1$.

Table 4
A comparison of the combined recall levels of SS, US, TTL, and PHIRST methods in 1000 LL, LM, LH, MM, MH, and HH artificial queries

	SS	US	TTL = 100	PHIRST
LL	3	3	0	3
LM	68	68	2	68
LH	1167	1167	47	1167
MM	874	874	93	870
MH	4626	4626	1180	4626
HH	5000	5000	4997	5000

element of the Hybrid Algorithm 4 in lines 1 and 2. However, unless extreme changes in the document density occur (e.g. every node contains a large percentage of the documents), differences in the search costs are so large that this parameter is unlikely to have any impact on which algorithm should be used in categories such as LL, LM, and LH. The structured approach is completely unaffected by document density, and thus the Hybrid's structured component is uninfluenced as well.

The results in Table 4 display the combined number of query results (recall) returned from each search algorithm with the maximal results set here to 5 ($T = 5$) and 1000 queries. This result underlies the potential strengths and weakness of the PHIRST method. Despite the lower costs of PHIRST, this approach was overall equally effective in returning the query results. When word combinations were frequent, the US component of the PHIRST method still found the results (thus MH was still successful). At the other extreme, assuming the word frequency of any term was less than d, at least one term was fully indexed. In these cases, complete recall was also guaranteed if SS was used on the indexed term(s) followed by the US to find all remaining terms. In addition, all terms taken from the L category were in less than d documents (as L values had 50 or fewer instances while $d = 75$), resulting in full recall for all of these categories (LL, LM, and LH). As predicted in Section 5, the query algorithms did experience minor difficulties in finding series of terms of medium frequency. Note that the PHIRST method did return slightly fewer results in the MM cases (870 versus 874).

We found that this limitation was negligible in answering real-world queries once d was significantly higher than T. To verify this finding we used the 1000 most popular real movie keywords taken from the Internet Movie Database[2] retrieved on October 25, 2006. These queries were typically between 1 and 4 words (mean 1.94).

Table 5 shows a comparison of the number of results found from these queries with the SS, US, and TTL = 100 methods, and the PHIRST method with $d = 75$ and variable values for T. The results from the SS and US algorithms represent baseline algorithms that found the maximal number of results (100% recall) for the 1000 queries. For example, with $T = 5$, a total of 4592 combined hits were found given these queries. Both the TTL = 100 and PHIRST algorithms did not guarantee 100% recall, albeit with markedly lower search costs. Note that the PHIRST algorithm found nearly all results (99.89% of the results found by the complete US and SS algorithms) when only 5 results were requested ($T = 5$). PHIRST held up fairly well even when 20 matches ($T = 20$) were required with 97.78% of all matches found. The recall of the PHIRST approach dropped with T (92.77% at $T = 50$, and only 33.23% at $T = D$). This confirms the claim that in real

[2] (http://www.imdb.com/Search/keywords)

Table 5
A comparison of the combined recall levels of SS, US, TTL, and PHIRST methods with reference to different numbers of results (T) and 1000 queries

	SS	US	TTL = 100	PHIRST
$T = 5$	4592	4592	2138	4587
$T = 20$	15 598	15 598	3712	15 252
$T = 50$	30 347	30 347	4534	28 154
$T = 2000$	105 649	105 649	5254	35 087

Table 6
A comparison of the cost levels of SS, US, TTL, and PHIRST methods with reference to different numbers of results (T)

	SS	US	TTL = 100	PHIRST
$T = 5$	57 680	591 841	86 578	12 006
$T = 20$	68 696	1 181 515	97 735	24 976
$T = 50$	83 435	1 567 039	99 269	38 744
$T = 2000$	158 737	2 000 000	100 000	68 610

queries the recall of the PHIRST approach would be nearly 100% for $T \ll d$ (e.g., $T = 5$), but would perform poorly once $T \gg d$ (e.g., $T = D$). In comparison, the TTL = 100 algorithm performed much worse, even in the case of $T = 5$ with only 2138 total results found.

Table 6 displays the search costs for executing these real queries within the four algorithms described in this paper assuming $C_S = C_U = 1$, and each agent stores only one document. We again found that the PHIRST approach had significantly lower search costs than all three of the other approaches. Again, observe that the advantage of the PHIRST approach was most evident when $d \gg T$. If $T = 5$, the PHIRST approach incurred a cost of nearly $\frac{1}{5}$ the cost of the next best method (SS) (with a high recall of 99.89%). If $T = 20$, its cost, nevertheless, was nearly $\frac{1}{3}$ that of the next best method (SS) (still with a high recall of 97.78%). If $T = D$, the cost advantage of the PHIRST approach was under $\frac{1}{2}$ of the next best method (TTL = 100) (the recall was only 33.23%).

7.3. Churn experiments

Recall that the PHIRST approach to handling churn requires that k copies of each inverted list must be stored. We conducted experiments studying the relationship between the value of k, the system's publishing requirements in creating these k copies, and the search costs related to executing queries. The goal was to achieve at least 95% of the query results when confronted with churn compared to the results achieved when no churn existed while minimizing publishing and search costs.

We simulated conditions with k set at 1, 5, and 15 and studied their impact on the publishing storage and query results of the Hybrid algorithm and set $d = 75$ for the publishing algorithm. We revisited the real-world queries from the previous dataset, again assumed 2000 nodes published a total of 2000 documents, and studied the case where the goal was to return 20 matches (T), and a scenario with a very high churn rate of 0.5. To simulate churn, we created random snapshots of the simulator where half of the nodes were chosen at random to have failed with uniform distribution. In the first set of experiments we assumed that when the entire set of k nodes were "down" the query would fail. The results from this experiment are presented in Table 7.

As these results indicate, there is a clear tradeoff between having additional nodes within k, their storage requirements, and the query results. Setting $k = 1$ had the lowest publishing load, but also meant that each term had no replicated copies. Note that this value of 134.07 is identical to the result found in the top portion of Fig. 2 for the data point where the number of nodes is 2000. Keeping additional copies of inverted term data increased the published load per node proportionately. As a result, one could publish one redundant dataset ($k = 2$) in PHIRST and still incur a cheaper publishing cost than naively publishing all terms (PHIRST $k = 2$ encountered a publishing load cost of 268.14 compared to the naive publishing load of 315.67 seen in Fig. 2). While setting $k = 15$ resulted in the highest average published load, it allowed the algorithm to find all possible results (see query results in Table 7). In this experiment, we assumed a query would fail if a SS was desired, but the node with the inverted list(s) had failed. As a result, note that the numbers in the third column (search costs in Table 7) increase as k increases. In the case of $k = 1$, the search failed many times, thus resulting in significantly lower search costs in conjunction with the lower query results. As predicted mathematically (see end of Section 6), setting $k = 5$ results in an effective tradeoff between achieving over 95% recall from the inverted lists (96% of the results from $k = 15$) while still keeping the publishing load relatively low ($\frac{1}{3}$ of the publishing load of $k = 15$).

Next, we repeated the above experiment, but assumed that an US would be used if the node with the inverted list failed. These results are presented in Table 8. Note that the publishing costs (column 2 in Table 8) are identical to those depicted in Table 7. However, as opposed to the results in Table 8, here the query results remained fairly constant. Nonetheless, note that in this experiment the search cost for $k = 15$ was the lowest since the entire k set of nodes never failed, and thus random search was never used. Conversely, setting $k = 1$ resulted in often using the random search, which here caused the highest search cost. Also, small fluctuations existed in the query results, with performance slightly decreasing as k increased. First, as

Table 7
A comparison of the impact of redundant nodes on the publishing load, query results, and search cost within the Hybrid method with $d = 75$ and $T = 20$ when node failure results in search termination

Replicated nodes (k)	Publishing load	Query results	Search costs
1	134.07	6756	6015
5	670.35	14 692	24 308
15	2011.05	15 272	24 919

Table 8
A comparison of the impact of redundant nodes on publishing load, query results, and search costs of the Hybrid method where $d = 75$ and $T = 20$ when node failure results in an unstructured search

Replicated nodes (k)	Publishing load	Query results	Search costs
1	134.07	15 428	566 477
5	670.35	15 259	59 994
15	2011.05	15 277	35 544

each experiment randomly decided which nodes should fail, slight differences existed between trials. Additionally, when a node's inverted list was unavailable, as often occurred when $k = 1$, a full US was used. Despite the high search costs, this approach did infrequently find results that the PHIRST hybrid approach would have missed as SS had been desired, but not enough entries had been indexed to return the full 20 results. Referring back to row 2 of Table 5 (results of $T = 20$) we see that the full SS found 15 598 matches in the equivalent experiment without churn, as opposed to 15 252 found by PHIRST. However, note that these differences represent less than 1% of the highest query result value (when $k = 1$). Finally, here $k = 5$ again provided a good tradeoff between publishing costs, query results, and search costs.

Based on these results we concluded that the PHIRST approach was successful in reducing the publishing load, even in systems with a very high churn rate.

8. Conclusions

In this work we have presented PHIRST, the first system capable of executing distributed P2P full-text search. PHIRST contains novel publishing algorithms that ensure that no agent will be required to store more than d entries in its inverted list of a given term. This allows PHIRST's publishing algorithms to partially index all words in the corpus and still keep the storage costs allocated equitably. More importantly, this approach also makes PHIRST highly scalable since the average amount of the inverted file information actually decreases as the number of agents and documents in the system increases. We have also presented query algorithms that select the best search approach based on global frequencies of all words in the corpus. These algorithms allow PHIRST to choose the best method based on estimated costs. PHIRST uses US to effectively compensate for the lack of inverted lists of terms published and SS to locate rare terms. Finally, we have shown that PHIRST can handle issues related to both scheduled and unscheduled node failures.

References

[1] J. Li, B. Loo, J. Hellerstein, F. Kaashoek, D. Karger, R. Morris, On the feasibility of peer-to-peer web indexing and search, in: Second International Workshop on Peer-to-Peer Systems, 2003.
[2] ⟨www.google.com⟩.
[3] P. Brighten Godfrey, S. Shenker, I. Stoica, Minimizing churn in distributed systems, in: SIGCOMM '06: Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, 2006, pp. 147–158.
[4] S. Rhea, D. Geels, T. Roscoe, J. Kubiatowicz, Handling churn in a DHT, in: Proceedings of the 2004 USENIX Annual Technical Conference (USENIX '04), 2004.
[5] Y. Yang, R. Dunlap, M. Rexroad, B.F. Cooper, Performance of full text search in structured and unstructured peer-to-peer systems, in: IEEE INFOCOM, 2006, pp. 2658–2669.
[6] L. Gravano, H. García-Molina, A. Tomasic, Gloss: text-source discovery over the internet, ACM Trans. Database Syst. 24 (2) (1999) 229–264.
[7] B. Pang, L. Lee, S. Vaithyanathan, Thumbs up?: sentiment classification using machine learning techniques, in: EMNLP '02: Proceedings of the ACL-02, 2002, pp. 79–86.
[8] ⟨www.imdb.com⟩.
[9] T. Joachims, A probabilistic analysis of the Rocchio algorithm with TFIDF for text categorization, in: Proceedings of ICML-97, 14th International Conference on Machine Learning, 1997, pp .143–151.
[10] R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan, Chord: a scalable peer-to-peer lookup service for internet applications, in: ACM SIGCOMM 2001, 2001.
[11] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, J.D. Kubiatowicz, Tapestry: a resilient global-scale overlay for service deployment, IEEE J. Selected Areas Commun. 22 (1) (2004) 41–53.
[12] P. Reynolds, A. Vahdat, Efficient peer-to-peer keyword searching, in: Middleware, 2003, pp. 21–40.
[13] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, S. Shenker, Making gnutella-like p2p systems scalable, in: SIGCOMM '03: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, 2003, pp. 407–418.
[14] Q. Lv, P. Cao, E. Cohen, K. Li, S. Shenker, Search and replication in unstructured peer-to-peer networks, in: ICS '02: Proceedings of the 16th International Conference on Supercomputing, 2002, pp. 84–95.
[15] B.T. Loo, J.M. Hellerstein, R. Huebsch, S. Shenker, I. Stoica, Enhancing p2p file-sharing with an internet-scale query processor, in: Proceedings of VLDB, 2004, pp. 432–443.
[16] B.T. Loo, R. Huebsch, I. Stoica, J.M. Hellerstein, The case for a hybrid p2p search infrastructure. in: Proceedings of IPTPS, 2004, pp. 141–150.
[17] A. Rosenfeld, C.V. Goldman, G.A. Kaminka, S. Kraus, An architecture for hybrid p2p free-text search, in: Cooperative Information Agents XI, Lecture Notes in Computer Science, vol. 4676, Springer, Berlin, 2007, pp. 57–71.
[18] Y.-J. Joung, C.-T. Fang, L.-W. Yang, Keyword search in dht-based peer-to-peer networks, in: ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05), 2005, pp. 339–348.
[19] P.A. Bernstein, V. Hadzilacos, N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley Longman Publishing Co. Inc., Reading, MA, 1986.
[20] C.D. Paice, Another stemmer, SIGIR Forum 24 (3) (1990) 56–61.