

An Architecture for Hybrid P2P Free-Text Search

Avi Rosenfeld^{1,2}, Claudia V. Goldman³, Gal A. Kaminka², and Sarit Kraus²

¹ Department of Industrial Engineering

Jerusalem College of Technology, Jerusalem, Israel

² Department of Computer Science Bar Ilan University, Ramat Gan, Israel

³ Samsung Telecom Research Israel, Herzliya, Israel

Email: {rosenfa, galk, sarit}@cs.biu.ac.il, c.goldman@samsung.com

Abstract. Recent advances in peer to peer (P2P) search algorithms have 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 networks. 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 searches to compensate for the lack of fully published terms. Additionally, they explicitly weigh between the costs involved with structured and unstructured approaches, allowing for a significant reduction in query costs. 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 searching, or the ability to locate documents based on terms found within documents, is arguably one of the most essential tasks in any distributed network [5]. Search engines such as Google [16] 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 10 billion pages [16].

In this paper, we address the challenge of implementing full-text searches within peer-to-peer (P2P) networks. Our motivation is to demonstrate the feasibility of implementing a P2P network 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 cost 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 the amount of storage it has. 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.

To date, three basic approaches have been proposed for full-text searches within P2P networks [15]. Structured approaches are based on classic Information Retrieval theory [2], 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 not successful in locating hard to find items [15]

In this paper we present PHIRST, a system for **Peer-to-Peer Hybrid Restricted Search for Text**. PHIRST is a hybrid approach that leverages the advantages of structured and unstructured search algorithms. Similar to structured approaches, agents publish terms within their documents as they join or add documents to the P2P network. This information is necessary to successfully locate hard-to-find items. Unstructured search is used to effectively find common terms without expensive lookups of inverted lists. Another key feature in PHIRST is its ability to restrict the number of peer addresses stored within inverted lists. Not only does this insure that the hardware limitations of agent nodes are not exceeded, it also better distributes the system's storage. We also present a full-text query algorithm where nodes explicitly reason based on estimated search costs about which search approach to use, reducing query costs as well.

To validate the effectiveness of PHIRST, we used a real web corpus [11]. 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 [5, 15]. 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 term pairs 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 (www.imdb.com) and found PHIRST was in fact effective in answering these queries.

2 Related Work

Classical Information Retrieval (IR) systems use a centralized server to store inverted lists of every document within the system [2]. 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 [3]. 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 needing 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 [13], Chord [9], and Tapestry [14]. 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 through 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 $\text{Log}N$ hops, with N being the total number of agents in the network [9, 13]. 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 $\text{Log}N$ hops [9, 13]. Then, the agent must retrieve these lists and intersect them to find which peer(s) contain all of the terms.

Li et al. [5] 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 contain N entries. Thus, the storage requirements for these inverted lists are likely to exceed the hardware abilities of agents in these systems. 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 not feasible.

Several recent developments have been suggested to make a full text distributed system viable. One suggestion is to process the structured search 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 [15]. 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.

Unstructured search protocols provide an alternative that is used within Gnutella and other P2P networks [1]. These protocols have no publishing requirements. To find a document, the searching query 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 [8]. Furthermore, one can initiate multiple simultaneous "walks" to find items more quickly, or use state-keeping to prevent "walkers" from revisiting the same nodes [8]. Despite these optimizations, unstructured searches have been found to be unsuccessful in finding rare terms [1].

In super-peer networks, certain agents store an inverted list for all peer documents for which it assumes responsibility. Instead of publishing copies 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 its super-peers, only one hop is needed to publish a message, instead of the LogN 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 [15] – something we could not assume within our system.

Hybrid architectures involve using elements from multiple approaches. Loo et al. [6, 7] 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 unstructured search 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 DHT's instead of a super-peer system. Second, in determining if a file was common or not, they needed to rely on locally available information from super-peers, and used a variety of heuristics to attempt to extrapolate this information for the global network [6]. As 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. [7] only published the files' names, and not their content. As they considered full text search to be infeasible for the reasons previously presented [5], their system was limited to performing searches based on the data's file name, and not the text within that data. As our next section details, we present a publishing algorithm that actually becomes cheaper to use as subsequent nodes are added. Thus, PHIRST is the first system to facilitate effective full-text search even within large P2P networks.

3 PHIRST Overview

First, we present an overview of the PHIRST system and how its publishing and query algorithms interconnect. While this section describes how information is published within the Chord DHT [9], PHIRST's publishing algorithm is generally presented in section 4 so it may be used within other DHT's as well. Similarly, section 5 presents a query algorithm (algorithm 2) which generally selects the best search algorithm based on the estimated cost of performing the search algorithms at the user's disposal. The selection algorithm is generically written such that new search algorithms can be introduced without affecting the algorithm's structure. Only later, in algorithm 3 do we present how these costs are calculated specific to the DHT and unstructured search algorithms we used.

In order to facilitate structured full-text search for even infrequent words, search keys 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 key is found by using a preselected hash function, such as SHA-1, to compute the hash value of that key. Chord then routes the key to the agent whose Chord identifier is equal to or is the successor (the next existent node) of that value [9]. For example, Figure 1 is a simple example

with an identifier space of 8, and 3 nodes. Assuming the key hashes to a value of 6, that key needs to be stored on the next node within the circular space, or node 0. Assuming the key hashes to 1, it is stored on node 1.

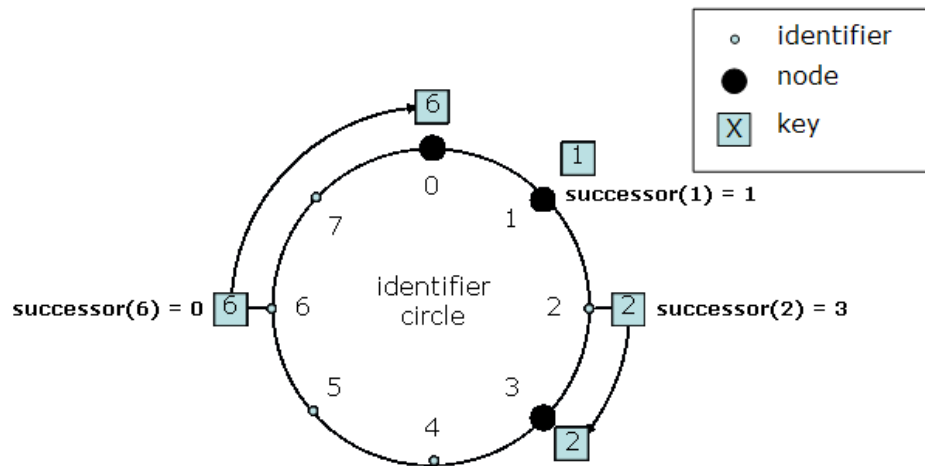


Fig. 1. An example of a Chord ring with $m=3$. Figure based on Chord paper [9].

The hashing quality within the Chord algorithm has several important qualities. First, it creates important performance guarantees, such as $\text{Log}N$ average search length. 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 persistent hashing function used by Chord has the quality that no agent will get more than $O(\text{Log}N)$ keys than the average [9]. We refer the reader to the Chord paper for further details [9].

However, the DHT's performance guarantees only balancing the number of keys stored per node, but not the number of addresses stored in the inverted lists for each key. For example, Table 1, gives an example of the inverted lists for five words. Common words, such as "a" and "the" within the table, will produce much long inverted lists, than uncommon words such as "aardvark" and "zygote". Due to space restrictions we will only present up to the first 7 inverted entries for each word, out of a potential length of N rows. Balancing guarantees only apply to the number of words (out of N), but not the size of each inverted list (the length of that row). Because word distribution within documents typically follow Zipf's law, some of the words within documents occur very frequently while many others occur rarely [4]. 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. Thus, one key contribution of this paper is a publishing algorithm that can equitable distribute these entries by allowing agents to cap the number of inverted list entries they will store.

Once the publishing stage has been begun, a distributed database exists to search the network for full-text queries. We define the search task as finding a number of results,

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

Word (key)	Address1	Address2	Address3	Address4	Address5	Address6	Address7
a	111-1111	111-1112	111-1113	111-1114	111-1115	111-1116	111-1117
aardvark	111-4323						
the	111-1111	111-1112	111-1113	111-1114	111-1115	111-1116	111-1117
zoo	123-4214	123-9714	333-9714				
zygote	548-4342						

T, that match all query terms within the documents' text. Capping a query at T results is needed within unstructured searches, as there is no global mechanism for knowing the total number of matches [15]. Finding only a limited number of results has also been previously suggested within structured searches to reduce communication costs [12]. The second key contribution of this paper is a novel querying algorithm that leverages between structured and unstructured searches to effectively find matches despite the limit in the amount of data each peer stores.

4 The Publishing Algorithm

Every time an agent joins the network, or an existing agent wishes to add a new document, it must publish the words in its document(s) as described in Algorithm 4. First, the agent generates a set of max terms it wishes to add (line 1). Similar to other studies [15] 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 [15]. Furthermore, as we detail in the Experimental Results section (section 6), stemming also further reduces the amount of information needed to be published and stored. The publishing agent, ID_{Source} , then sends every unique term, $Term_i$, to be stored in an inverted list on peer ID_{DEST} (lines 3-4). The keys being stored are these words that are sent, with each word either creating a new inverted list, or being added to an existing file. In addition to these terms, the agent also updates a counter of the total number of documents contained between all agents within the system (line 4). For simplicity, let us assume this global counter is stored on the first agent, ID_1 . We will see that this value is needed by the query algorithm described below.

Algorithm 1 Publishing Algorithm(Document Doc)

```

1: Terms  $\Leftarrow$  Preprocessed words in Doc
2: for  $i = Term_1$  to  $Term_{max}$  do
3:   PUBLISH( $Term_i, ID_{Source}, ID_{DEST}$ )
4: PUBLISH(DOC-COUNTER+1,  $ID_1$ )
5: for  $i = Term_1$  to  $Term_{received}$  do
6:   if SIZE( $ID_{DEST}, Term_i$ ) <  $d$  then
7:     ADD-Term( $Term_i, ID_{Source}$ )
8:   UPDATE-Counter( $Term_i, COUNTER$ )

```

PHIRST's publishing algorithm enforces an equitable term distribution by only storing inverted lists until a length of d . For every term node, $Term_i$ out of a total of

received terms, ID_{DEST} is requested to store it must decide if it should fulfill that request. As lines 6 and 7 of the algorithm detail, assuming agent ID_{DEST} currently has fewer than d entries for $Term_i$, it adds the value ID_{Source} to its list (or creates an inverted list if this is the first occurrence). Either way, nodes log that a certain number of $COUNTER$ instances of that term exist (lines 8). This information is used by the query algorithm to determine the global frequency of this term. Because we limit each node to only storing d out of a possible N terms, the storage requirements of the system are reduced to $d*N$ from $N*N$. As we set $d \ll N$, we found this savings to be quite significant.

Theoretically, additional information about each term may be published, such as the position that term occurred or how many instances of that term existed within the document and aggregate this and similar information into a rating for the term it is about to publish. This information may be especially important when more than d instances of that 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 top d highest rating. In a similar vein, if more than d instances of $Term_i$ exist, it may be advantageous to store the d most recent documents, especially if turnover exists within nodes.

The performance guarantees of DHT's such as Chord insure the publishing algorithm runs with fairly low cost. Because each node, ID_{Source} , needs $\log N$ hops to find the agent, ID_{DEST} , responsible for storing that term's inverted list, the total number of messages needed to publish a document is of order $O(max * \log N)$ where max is the number of terms in that document. Note that the publishing algorithm described here sends all terms, even those which in fact do not need publishing because they already contained d terms.

5 The Search Algorithm

The search algorithm is called once any agent wishes to conduct a distributed full-text search. As Algorithm 2 describes, this process operates in two stages. First, we retrieve the global frequencies of all search terms (line 1) and sort all terms from least to most frequent (line 2). This value can be calculated through looking up the frequency of that term ($COUNTER$), and dividing this number by the total number of documents ($DOC - COUNTER$). Finding these values requires one lookup of the value of $DOC - COUNTER$ (assumed to be stored on agent ID_1 in the publishing algorithm), as well as a lookup for the frequencies of each term from the agent storing term $Term_i$. Referring back to algorithm 4 note that the peer storing $Term_i$ has a counter with this value even if more than d instances of this term occurred.

Once the frequency 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 3). If unstructured search is deemed less costly, all terms are immediately searched for simultaneously (lines 7–10). This type of search can either terminate because T matches have been found or the search space has been exhaustively searched. If structured search is deemed less costly, that term's inverted list is requested, and the search space is intersected with that of the new term (line 12). Assuming we have reached the last term (lines 12-17) we return the first T matches found

Algorithm 2 Hybrid Search Algorithm(String $Query_1 \dots Query_{max}$)

```

1: space  $\leftarrow \infty$  {Used for initialization to all P2P nodes}
2: Retrieve Frequencies of  $Query_1 \dots Query_{max}$ 
3: Term  $\leftarrow$  Sorted Query Terms Least to most Frequent {Term is an array}
4: for  $i = Term_1$  to  $Term_{max}$  do
5:   Frequency  $\leftarrow$  Product of Frequencies( $Term_i \dots Term_{max}$ )
6:   Tradeoff  $\leftarrow$  Calculate-Tradeoff(space,  $Term_i \dots Term_{max}$ , Frequency)
7:   if Tradeoff > 0 then
8:     while Found < T AND NOT Exhausted(space) do
9:       Search-Unstruct(space,  $Term_i \dots Term_{max}$ )
10:    Break
11:  else
12:    space  $\leftarrow$  List( $Term_i$ )  $\cap$  space
13:    if  $i=Term_{max}$  then
14:      if space > T then
15:        return first T list entries
16:      else
17:        return all list entries

```

after all terms were successfully intersected. Once the structured search identifies that fewer matches than T matches were found (line 15) it returns all list entries (line 17).

This algorithm has several key features. First, the search process is begun starting with the least frequent term. This is done following previous approaches [15] to save on communication costs. We denote the inverted list length of the **least** common search term as $length(Term_1)$ where $length$ is a function that returns the size of an inverted list and $Term_1$ is the first term after the terms are sorted based on frequency. Each successive peer receives the previously intersected list, and locally intersects this information with that of its term (line 13). The result of this process is that intersected lists become progressively smaller (or at worse case stay the same size) with the maximum information any peer can send being bounded by $length(Term_1)$. 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. This is done because the information gained from this frequency information, such as bounding search costs to the size of the least frequent term, far outweighs the search costs involved with processing the query in two stages. Finally, as the search goal is to return T results, the last node within a structured search does not need to return its entire inverted list. Instead, it only needs to send the first T results (or failure or NULL as in line 17 if under T results exist). Because of this, the maximal search cost will be of order $(max-1) * length(Term_1) + T$ where max is the number of terms in the search query.

Arguably the most important feature of this algorithm is its ability to switch between using structured and unstructured searches midway through processing the query terms. Even if structured search is used for the first term(s), the algorithm iteratively calls the tradeoff algorithm (algorithm 3) after each term. Once the algorithm notes that unstructured search is cheaper, it immediately uses this approach 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 use unstructured search within f to find the remaining common words.

Similarly, note that this approach lacks a TTL (Time To Live) for its unstructured search. We assume unstructured searches are to be used only when the expected cost of using an unstructured search is low (see algorithm 3 below). We expect this to occur when the unstructured search will terminate quickly, such as when: (i) the search terms are very common from the onset or (ii) unstructured search is used to find the remaining common terms after structured search generated an inverted list of f terms.

We now turn to the search specific mechanism needed to identify which search types will have the higher expected cost. This tradeoff depends on T , or the number of search terms wanted, the costs specific to using the different types of searches, and d or the maximal number of inverted list entries published for each term. Algorithm 3 details this process as follows:

Algorithm 3 Calculate-Tradeoff(Space, $Term_i \dots Term_{num}$, Frequency)

```

1: Expect-Visit  $\leftarrow T / \text{Frequency}$  {Number of nodes Unstructured search will likely visit}
2: COSTS  $\leftarrow C_U * (\text{Expect-Visit}) - C_S * (\text{Sending}(\text{query-terms}))$ 
3: if COSTS > 0 then
4:   RETURN 1 {pure unstructured search}
5: else if COSTS < 0 AND Size( $Term_i$ ) < d then
6:   RETURN -1 {pure structured search for this term}
7: else
8:   space  $\leftarrow \text{List}(Term_i) \cap \text{space}$ 
9:   RETURN 1 {Use unstructured afterwards because of lack of more values}

```

First, the algorithm calculates the expected cost of conducting an unstructured search. The expected number of documents that will be visited in an unstructured search before finding T results is: $T / \text{term-frequency}$ (line 1). For example, if we wish to find 20 results, and the frequency of the term(s) is 0.5, this search is expected to visit 40 documents before terminating. We can compare this value to that of using a structured search, 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 unstructured search on one peer. We also assume that some cost C_S is associated with sending one entry from the inverted list (line 2). Because the cost of unstructured search is $C_U * T / \text{Frequency}$, and the cost of structured search is bounded by $C_S * ((max-1) * \text{length}(Term_i) + T)$, the algorithm can compare the expected cost of both searches before deciding how to proceed (lines 3-6).

For many cases, a clear choice exists for which search algorithm to use. Let us assume that $C_U = C_S = 1$, and assume that all documents have been indexed, or $d = DOC - COUNTER$. When searching for common words, the cost of using the unstructured search is likely to be approximately T . Processing the same query with structured search will be approximately the number of documents ($DOC - COUNTER$) or a number much larger than T . Conversely, for infrequent terms, say with one term occurring only T times, the cost of an unstructured search will be $DOC - COUNTER$ or a number much larger than T , while the structured search will only cost a maximum

of $T * max-1 + T$. Finally, structured search is also the clear choice for queries involving one term. Note that in these cases, no inverted lists need to be sent ($max-1=0$), and only the first T terms are returned. The cost of using unstructured search will be greater than this amount (except for the trivial case where the frequency of the term is 1.0).

There are two reasons why the most challenging cases involve queries with terms of medium frequency. In these cases, the cost of using both the structured and unstructured searches are likely to be similar. However, 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. As a result, the PHIRST approach is most likely to deviate from the optimal choice in these types of cases.

A second challenge results from the fact that we only published up to d instances of a given term. In cases where inverted lists were published without limitation, e.g. d equals N ($DOC - COUNTER$), the second algorithm contains only two possible outcomes – either the expected cost is larger for using structured search, or it is not. However, our assumption is that hardware limitations prevent storing this number of terms, and d must be set much lower than N . As a result, 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 $d+\epsilon$. While other options may be possible, in these cases our algorithm (in lines 7-9) takes the d terms from the inverted lists, and conducts an unstructured search for all remaining terms. In general, we found this approach will be effective so long as the $T < d$, or the relationship, $T < d \ll N$ exists. We further explore the impact of this limitation in the next section.

6 Experimental Results

In this section we present 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 [11]. 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 frequency of words. This experiment demonstrated the theoretical strengths and weaknesses of PHIRST. We also studied real movie queries based on the Internet Movie Database (www.imdb.com). These experiments demonstrated that any weakness in PHIRST is likely to be insignificant in handling real queries.

6.1 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. Figure 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 [11]. In the left graph, we used the Paice stemming algorithm [10] on each term before storing it. The right graph published each term without stemming. In both graphs we also ran the publishing algorithm with $d=25$ and 75.

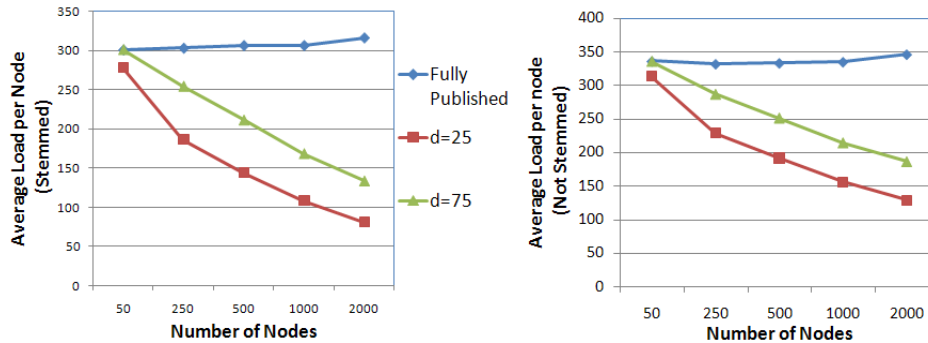


Fig. 2. Comparing publishing requirements of full publishing versus publishing limited to $d=75$.

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 $d=N$, 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 N grows. This results in progressively more words exceeding the d threshold, and no longer needing 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 networks.

Table 2. Average number of inverted list entries if 1 document published for every 2 agents.

Number of Nodes	50	250	500	1000	2000
Fully Published	150.43	151.51	153.13	153.1265	157.8343
d=25	138.84	93.106	72.17	53.97	40.605
d=75	150.43	127.14	105.72	84.38	67.035

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 Figure 2 * 0.5.

We also found a Zipfian distribution of terms with a long tail of infrequent terms (see Figure 3). Similar distributions have been found in P2P systems for items such as file frequency [6, 7] and term frequency [4]. The storage saving results we found were

from words with frequencies greater than d , or those terms towards the head of this distribution.

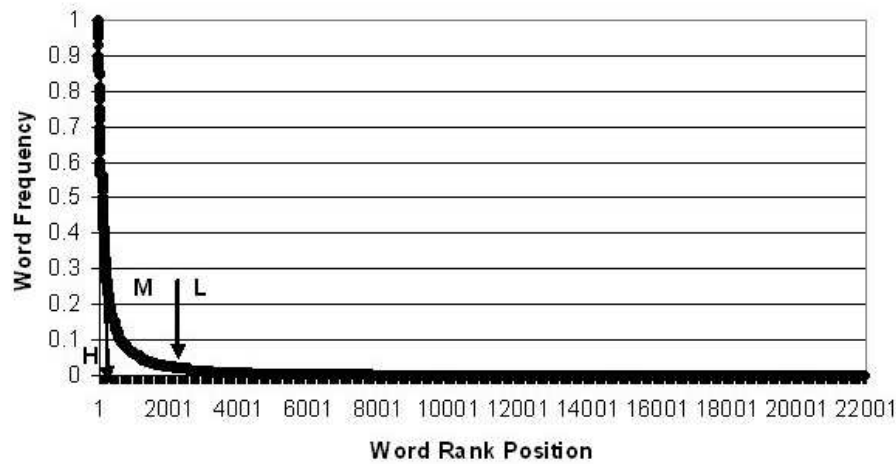


Fig. 3. Distribution of words by rank order within a movie corpus.

6.2 Query Experiments

We first conducted query experiments based on artificial queries chosen based on term frequency. Figure 3 displays the rank order of all words within the 2000 word corpus. 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 define 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 assume medium frequency words are those between these extremes.

We created paired terms (2 terms) of all permutations of these categories. This involves words both with high frequency (HH), both of low frequency (LL), both of 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 impact the query algorithm as terms are first sorted by the query algorithm 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 4 search algorithms. The Structured Search (SS) method published all terms and sent these indices between agents as necessary during queries. The Unstructured Search (US) used no publishing and used a random walk approach to find query results. The TTL=100 method used an unstructured search, 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. Comparing 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	2000000	100000	1466
LM	2206	2000000	100000	2142
LH	3177	1987754	100000	2010
MM	20732	1865474	99953	13256
MH	60188	234211	95624	18075
HH	871986	19746	20077	19995

Table 3 displays the average number of nodes visited (in the case of unstructured search) and / or the inverted list entries sent (for structured search) in finding 20 matches from each query ($T=20$). For simplicity, we assume that the costs of visiting nodes through unstructured search, and sending inverted list entries are equal, or $C_U = C_S$. As expected, we find that Structured Search (SS) is most expensive in finding common terms; where Unstructured Search (US) is most effective. Conversely, SS is most effective in finding rare terms. The hybrid PHIRST approach operates similarly to SS in finding rare terms (LL) and US in finding common items (HH). Note that in middle categories (for example MH) this approach sent the least amount of information. PHIRST saves costs by only sending a maximum of d entries even when structured search is deemed necessary. Furthermore, this approach switches between the SS and US methods as needed, saving additional costs.

The results in Table 4 display the number of query results returned from each search algorithm. This result underlies the potential strengths and weakness within 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 unstructured search component of the PHIRST method still found these 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 structured search is used on the indexed term(s) followed by unstructured search to find all remaining terms. In these experiments, all terms taken from the L category were in less than d documents (e.g. L values had 50 or fewer instances while $d=75$), resulting in full recall for all of these categories (LL, LM, and LH) as well. As predicted in section 5, the query algorithm did have slight trouble in finding series of terms of medium frequency. Note that the PHIRST method did return slightly fewer results in the MM case (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 claim we used the 1000 most popular real movie keywords taken from the Internet Movie Database Internet Movie Database⁴

⁴ (<http://www.imdb.com/Search/keywords>)

Table 4. Comparing recall levels of SS, US, TTL, and PHIRST methods in 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

taken from October 25, 2006. These queries were typically between 1 and 4 words (mean 1.94).

Table 5 compares the number of results found from these queries with SS, US, and TTL=100 methods, and the PHIRST method with $d=75$ with variable values for T . Note that the PHIRST algorithm found nearly all results (99.89%) 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 $T=N$). This confirms the claim that in real queries the recall of the PHIRST approach will be nearly 100% for $T \ll d$ (e.g., $T=5$), but performed poorly once $T \gg d$ (e.g., $T=N$).

Table 5. Comparing recall levels of SS, US, TTL, and PHIRST methods with regard to different numbers of results (T).

	SS	US	TTL=100	PHIRST
$T=5$	4592	4592	2138	4587
$T=20$	15598	15598	3712	15252
$T=50$	30347	30347	4534	28154
$T=2000$	105649	105649	5254	35087

Table 6 displays the search costs for finding these real queries for the 4 algorithms described in this paper assuming $C_S = C_U = 1$, and each agent stored only one document. We again found the PHIRST approach had significantly lower search costs than all three of the other approaches. Again, observe that the advantage to the PHIRST approach is most effective when $d \gg T$. If $T=5$, the PHIRST approach has nearly 1/5 the cost of the next best method (SS) (with a high recall of 99.89%). If $T=20$, its cost is still nearly 1/3 that of the next best method (SS) (recall still high at 97.78%). If $T=N$, the cost advantage of the PHIRST approach is under 1/2 from the next best method (TTL=100) (recall only 33.23%).

7 Conclusion

In this work we present PHIRST, a hybrid P2P search approach that leverages the strengths of structured and unstructured search. We present a P2P publishing algorithm that insures that no agent can hold more than d entries in its inverted list of a given term. This ensures that no one agent is required to hold disproportional amounts of data. PHIRST is highly scalable in that every agent typically stores fewer entries as

Table 6. Comparing cost levels of SS, US, TTL, and PHIRST methods with regard to different numbers of results (T).

	SS	US	TTL=100	PHIRST
T=5	57680	591841	86578	12006
T=20	68696	1181515	97735	24976
T=50	83435	1567039	99269	38744
T=2000	158737	2000000	100000	68610

the number of agents grows. This allows us to partially index all words in the corpus while keeping storage costs low. We also present a querying algorithm that selects the best search approach based on global frequencies of all words in the corpus. This allows us to choose the best method based on estimated cost. PHIRST uses unstructured search to compensate for the lack of published inverted list terms and structured search to location rare terms.

References

1. Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and N. Shenker. Making gnutella-like p2p systems scalable. In *SIGCOMM '03*, pages 407–418, 2003.
2. L. Gravano, H. García-Molina, and A. Tomasic. Gloss: text-source discovery over the internet. *ACM Trans. Database Syst.*, 24(2):229–264, 1999.
3. T. Joachims. A probabilistic analysis of the Rocchio algorithm with TFIDF for text categorization. In *Proceedings of ICML-97*, pages 143–151, 1997.
4. Y. Joung, C. Fang, and L. Yang. Keyword search in dht-based peer-to-peer networks. In *ICDCS '05*, pages 339–348, 2005.
5. J. Li, B. T. Loo, J. M. Hellerstein, M. F. Kaashoek, D. R. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *Proc. IPTPS*, 2003.
6. B. T. Loo, J. M. Hellerstein, R. Huebsch, S. Shenker, and Ion Stoica. Enhancing p2p file-sharing with an internet-scale query processor. In *Proceedings of VLDB*, pages 432–443, 2004.
7. B. T. Loo, R. Huebsch, I. Stoica, and J. M. Hellerstein. The case for a hybrid p2p search infrastructure, In *Proc. IPTPS*, 2004.
8. Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS '02*, pages 84–95, 2002.
9. R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*, pages 149–160, 2001.
10. C. D. Paice. Another stemmer. *SIGIR Forum*, 24(3):56–61, 1990.
11. B. Pang, L. Lee, and S. Vaithyanathan. Thumbs up?: sentiment classification using machine learning techniques. In *EMNLP '02*, pages 79–86, 2002.
12. P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Middleware*, pages 21–40, 2003.
13. S. Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *USENIX '04*, pages 127-140, June 2004.
14. B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.
15. Y. Yang, R. Dunlap, M. Rexroad, and B. F. Cooper. Performance of full text search in structured and unstructured peer-to-peer systems. In *IEEE INFOCOM*, 2006.
16. www.google.com