SPT – Summary Prefix Tree: An over DHT Indexing Data Structure for Efficient Superset Search

Bassirou Ngom, Mesaac Makpangou, Samba Ndiaye

SPT – Summary Prefix Tree

An over DHT Indexing Data Structure for Efficient Superset Search

Bassirou Ngom [*+] — Mesaac Makpangou[+] — Samba Ndiaye[*]

[*] Département Mathématiques-Informatique
Université Cheikh Anta Diop
Dakar - SENEGAL
(bassirou83.ngom, samba.ndiaye)@ucad.edu.sn

[**] Sorbonne Universités, UPMC Univ Paris 06, CNRS
INRIA - LIP6
Paris - FRANCE
(bassirou.ngom, mesaac.makpangou)@lip6.fr

RÉSUMÉ. Cet article présente un arbre de préfixes SPT, une structure de données qui permet de réaliser efficacement des recherches de sur-ensemble sur DHT. Chaque document est résumé par un filtre Bloom qui est ensuite utilisé par SPT pour indexer ce document. SPT implémente une procédure de recherche hybride qui est bien adaptée aux clés d'indexation éparses telles que les filtres Bloom. Nous proposons aussi une fonction de mapping qui atténue l'impact de l'asymétrie de SPT en raison de la rareté des bit 1 dans les filtres de Bloom, surtout lorsqu'ils ne contiennent que peu de mots. Pour effectuer des recherches de sur-ensemble, SPT maintient sur chaque noeud une vue locale de l'arbre global. Les principales contributions sont les suivantes. Premièrement, l'approximation de la relation de sur-ensemble entre les ensembles de mots-clés par la relation descendance entre les filtres Bloom. Deuxièmement, l'utilisation d'un arbre de préfixes (SPT), une structure d'indexation de données pour la recherche par mot-clé sur DHT. Troisièmement, une procédure de recherche hybride qui exploite la nature éparse des filtres Bloom pour offrir de bonnes performances. Enfin, un algorithme qui exploite SPT pour trouver efficacement des descriptions qui sont des sur-ensembles d'une requête de mots-clés.

ABSTRACT. This paper presents the summary prefix tree (SPT), a trie data structure that supports efficient superset searches over DHT. Each document is summarized by a Bloom filter which is then used by SPT to index this document. SPT implements an hybrid lookup procedure that is well-adapted to sparse indexing keys such as Bloom filters. We also propose a mapping function that permits to mitigate the impact of the skewness of SPT due to the sparsity of Bloom filters, especially when they contain only few words. To perform superset searches, SPT maintains on each node a local view of the global tree. The main contributions are the following. First, the approximation of the superset relationship among keyword-sets by the descendant relationship among Bloom filters. Second, the use of a summary prefix tree, a trie indexing data structure, for keyword-based search over DHT. Third, a hybrid lookup procedure which exploits the sparsity of Bloom filters to offer good performances. Finally, an algorithm that exploits SPT to efficiently find descriptions that are supersets of query keywords.

MOTS-CLÉS : SPT, recherche de sur-ensembles, indexation des données, DHT

KEYWORDS : SPT, Superset search, over-DHT, Data indexing

1. Introduction

Keyword-based search is an essential service for all users. On the one hand, many applications can benefit from a scalable, fault-tolerant, and robust distributed keyword-based searching system to improve information sharing among peers. For instance in many developing countries, universities do not have enough resources (e.g., bandwidth, servers) to make their scientific publications widely available online. These universities also have limited resources dedicated to accesses to digital libraries. As a consequence, most researchers and students of developing countries lack a common view of what research projects are undertaken on different universities, often leading to similar projects, which ignore each other. Also, scarce resources are often wasted to download multiple times documents that are available nearby. We argue that a nation-wide index of documents available on peers, that is scalable, fault-tolerant, robust and offering efficient keyword based search, can help improve the accessibility of scientific information to members of the community with no extra cost. Each peer simply registers the URLs and the abstracts of documents in her/his possession to make them accessible to the entire community.

On the other hand, Distributed Hash Table (DHT) is a widely used building block for scalable, fault-tolerant, and robust peer-to-peer systems. However, supporting efficient scalable keyword-based search on DHT is a challenge. Several proposals [1], [2], [3], [4], [5] that rely on a Distributed Inverted Index are confronted to a number of drawbacks, mainly high bandwidth consumption, uneven load of nodes, and the weak filtering of information when querying with popular keywords. Joung et al [6] propose to rely on an r dimension hypercube to build the index. While this proposal is interesting especially for exact searches, it exhibits poor performances for superset search. In this paper, we extend the use of a trie data structure to build an index supporting keyword-based search. While number of proposals [7], [8], [9], [10] rely on trie data structures to efficiently support complex queries over DHT, this approach is insufficiently investigated for keyword-based search over DHT. Firstly we summarize each document by a Bloom filter, and then transform the keyword-based search to the search of descendants of a query Bloom filter. Intuitively, some filter f is a descendant of a query Bloom filter q if both are built thanks to the same hash functions, have the same length, and for each position of q set to 1 the corresponding position of f is also set to 1. Secondly we build a summary prefix tree (SPT), a trie data structure for keyword-based search over DHT. SPT indexes records, each consisting of a 2-tuple (summary, URL) where URL refers to the location of the document. SPT implements a hybrid lookup procedure that is well adapted to sparse indexing keys such as Bloom filters.

We propose a mapping function that permits to mitigate the impact of the skewness of SPT and the high number of split operations. Finally, like LIGHT [8], SPT maintains on each node, a local view of the global tree and exploits these local views to conduct an efficient superset search. To sum up, this paper presents four main contributions. First, we approximate the superset relationship among keyword-sets by the descendant relationship among Bloom filters. Such a transformation permits to deal with compact set summaries, together with efficient bitwise operations. Second, the use of a summary prefix tree, a trie index data structure to build a scalable, fault-tolerant and robust over DHT index for keyword-based search. Third, a hybrid lookup procedure that suits the specific of sparse indexing keys. Unlike the traditional linear lookup that tests all prefixes until it reaches the one that identifies the appropriate leaf node, the hybrid procedure jumps directly to prefixes that are significant for superset search. This procedure is particularly efficient for

the location of sparse indexing keys. Finally, a superset search algorithm that exploits the proposed SPT trie data structure to efficiently retrieve satisfying documents. The rest of the paper is organized as follows. Section II presents existing related work. Then section III introduces the use of Bloom filter to approximate superset test. Section IV presents the summary prefix tree and its main operations, while section V discusses our superset search algorithm. Section VI concentrates on the performance evaluation. Finally, section VII draws some conclusions and points some perspectives.

2. Related works

Number of keyword-based searching systems [1], [2], [3], [4], [11] rely on a Distributed Inverted Index. Distributed Inverted Indices are confronted to a number of drawbacks : high bandwidth consumption, uneven load of nodes, and the weak filtering of information when querying with popular keywords. Keyword Fusion [5] is an inverted index that maintains a global dictionary of popular keywords. The efficient maintenance of the global dictionary is however a difficult challenge to deal with. To avoid the drawbacks inherent to Distributed Inverted Indices, Joung et al [6] propose to rely on an r ?dimension hypercube to build the index. Each document is associated a keyword set that characterizes its content. The system uses a hash function to summarize each description by an r-bits vector. Each document is indexed by the server associated with the r-bits that characterizes its description. Two types of search query are supported : pin search and superset search. A pin search finds the sets of entries for a given set of keywords. Superset search is conducted with a spanning tree. In this proposal, Bloom filters are used as a mean to group documents into separated clusters, while in our SPT proposal Bloom filters serve to approximate superset relationship. Hence SPT needs to build Bloom filters that guarantee a false positive rate lower than some predefined threshold. A second difference is that, rather than relying on an ad ?hoc indexing data structure such as a hypercube, SPT is an over-DHT indexing scheme that relies on a prefix tree. Mkey [12] is an overlay dependent indexing system. Mkey derives from each description summary a set of node identifiers, and then replicates this summary on the identified servers. To search items that match a query keyword set, Mkey determines the identifiers of servers that are responsibles of indexing items that match this request and send them the request. The result is obtained by making the union set of results returned by different indexing servers. While Mkey and SPT rely on Bloom filter to represent descriptions, SPT is an over-DHT while Mkey is an overlay dependent solution. A number of over-DHT index indexing schemes [7], [13], [8], [9] have been proposed to support complex queries over DHT. PHT [7] is an over-DHT index to support complex queries. PHT considers a bounded one-dimensional data space. Each item to index is represented by D-bits data key. PHT relies on a binary prefix tree, a trie data structure where all data are stored on leaf nodes. Each PHT node is uniquely identified by its label. PHT uses this label as the DHT key to store items assigned to the node. PHT supports split and merge operations. Splitting and merging require migrating data among DHT nodes. To insert (resp. delete) a key K into (resp. from) the prefix tree, one first locates the leaf node that covers K (i.e., a leaf node whose the label is a prefix of K). PHT proposes both a linear and binary solutions to lookup the appropriate prefix. Both solutions rely on DHT-lookup operations to check PHT node status. PHT maintains a double list that connects neighboring leaf nodes that permit to traverse the leaf nodes. Tang et al propose LIGHT [8], an over-DHT indexing scheme that supports efficient complex queries at a low maintenance cost. LIGHT relies on three novel

features namely : space partitioning tree, tree summarization, and naming function. The space partition tree recursively partitions the data space into two equal-size subspaces until each subspace contains fewer than θ records, where θ is the storage capacity of each leaf node. LIGHT associates a leaf bucket to each leaf node of the partition tree. A bucket data structure maintains a store that contains the records assigned to the corresponding leaf node, plus the label of that node. From a node label, one can infer its local view of the partition tree that consists of all the ancestors of the node and their siblings. The naming function is used to distribute the index structure over the DHT. LIGHT supports split and merge operations, though at a lower cost compared to PHT [7]. Thanks to its naming function, at split, one of the child nodes is mapped to the same DHT key as its parent, hence avoiding the migration of data assigned to that node. LIGHT relies on local trees to serve complex queries. Our SPT proposal is largely inspired by LIGHT. In particular, we use a naming function to map SPT leaf nodes into the DHT. We also adopt the tree summarization strategy.

3. Superset test approximation

Bloom filters are compact data structures used by many applications, in several distinct contexts, to approximate membership tests. Assuming that we have summarized two keyword sets A and B by Bloom filters of the same length, we want to use these summaries to decide for instance whether A contains B. Note that this can be done by repeating the membership test for each element of B. Unfortunately, in certain situations these elements are no longer at hand.

Let S_m designate the set of summaries of lenght m constructed thanks to the same set of hash functions. We define the relationship $descendant$ among the elements of S_m, noted $\hookrightarrow$, as follows : $\forall f, q \in S_m, f \hookrightarrow q \Leftrightarrow (q \wedge f) == q)$ where $\wedge$ is the bitwise intersection of bit strings.

Given two keyword-sets F_1 and F_2, we decide that F_1 contains F_2 if $(f_1 \hookrightarrow f_2)$ where f_1 and f_2 are the summaries of F_1 and F_2 respectively, of same lenght and constructed thanks to the same hash functions.

This decision is an approximation with a risk of a false positive decision (that is deciding that F_1 is a superset of F_2 while it is not true). If $nmax$ designates the maximum number of keywords in F_1 and F_2, the higher the ratio ($\frac{m}{nmax}$), the lower the risk for false positive decision for membership test approximations [14] and hence for the superset test. In this paper we assume that m is determined such as to ensure an acceptable false positive rate for superset tests.

Using the above supeset test approximation, we tranform the superset seach problem to the "summary descendant search problem". To address this new problem, we propose to index descriptions summaries and to offer efficient descendant search operations.

4. Summary Prefix Tree

Summary Prefix Tree (SPT) is an indexing data structure that index records. Each record is a 2-tuple, (*summary*, *docURI*), where summary is the Bloom filter that summarizes the description associated with the document identified by *docURI*.

SPT is a binary tree. By convention, the SPT root node is labelled "/". The left branch departing from any internal node is labelled 0 while the right one is labelled 1.

Any SPT node is uniquely identified by the string constituated of the root label concatenated with the labels of branches from the root through that node.

Each leaf node is associated a bucket that contains this node's label, a store of the set of records under the responsability of this node, and this node status (i.e., LEAF).

To distributed nodes buckets over a DHT, we define the function, $skey()$, that maps each SPT node identifier to a DHT storage key. Equation 1 sketches the mapping algorithm : nid is the node identifier to map to the DHT, z (resp. p) designates the longest prefix of nid with its rightmost bit equal to 0 (resp. 1), and $[0]*$ (resp. $[1]*$) refers to a sequence of bit 0 (resp. 1) repeted zero or more times.

$$skey(nid) = \begin{cases} "/", & if\ nid = "/" \\ "/0", & if\ nid = "/0[0]*" \\ "/1", & if\ nid = "/1[1]*" \\ "/p0", & if\ nid = "/p0[0]*" \\ "/z1", & if\ nid = "/z1[1]*" \end{cases} \quad (1)$$

To sum up, `skey()` maps any node identifier to the storage key obtained by replacing the longest rightmost bit sequence of identical value by one single bit of same value. For instance, the SPT nodes identified by "/10", "/100", "/1000", "/10000" and "/100000" are all mapped to the same DHT storage key "/10". Similarly "/01000111", "/010001111", "/0100011111" are mapped to the same DHT storage key which is "/010001".

4.1. SPT Construction and Maintenance

4.1.1. Insertion of New Summaries

At anytime, one can insert new records within the SPT data strucure. To insert a new record within SPT, one performs four operations (see Algorithm 1). Firstly, computes the indexing key associated with the new summary. Secondly, locates the leaf node to which this summary is assigned. Thirdly, maps the identifier of the node in charge of the new summary to its corresponding storage key. Finally, the new summary is stored in the DHT using the value returned by $skey()$ as DHT storage key.

Algorithm 1 SPT-insert(fb, docURI)

Require: fb, docURI : record to insert within the index
1: ikey ← computeIndexkey(fb)
2: nodeLabel ← SPT-lookup(ikey)
3: dhtKey = skey(nodeLabel);
4: DHT-put(dhtKey, (fb, docURI));

Similary to other related proposals the capacity of each SPT leaf node is limited to some system configuration variable B. When a leaf node reaches its maximum capacity, it splits into two leaf nodes, then the left branch is labelled "0", while the right one is labelled "1". Each child node is assigned a subset of the content the leaf node.

To split the content of a leaf node, the split procedure retrieves $label$ (contained within the bucket associated with this leaf node) then checks the bit at rank $label.length()$ of the index key for each summary contained in the associated bucket. If this bit is "0", the corresponding record is assigned to the left child, otherwise it is assigned to the right child.

Thanks to the specifics of our $skey()$ mapping function, the child node that has the same rightmost bit as its parent is mapped to the same storage key as its parent. Hence, after a node splits, apart for the case of root node, only a subset of content is migrated to a différent DHT storage node.

4.1.2. Removals of Indexed Documents

At anytime, one can request to remove an existing record from SPT. To do that, the system proceeds the same way as for insertion. Firstly, it computes the indexing key of the concerned summary ; secondly, it locates the leaf node in charge of that indexing key ; thirdly, it determines the storage key corresponding to the located node ; finally, it removes the concerned record from bucket stored within the DHT with that storage key.

The removal of a record can lead to a situation where the sum of records assigned to a node and its sibling falls under the maximal capacity of a leaf node. If this happens, the contents of these two siblings must be merged.

Algorithm 2 *mergeIfNeeded(node)* : merge node and its sibling if the size of the union of their contents is less than the maximun capacity of a leaf node

Require: $node$
1: **if** (node.content.size() $<$ $(B \div 2)$ **then**
2: sibling $\leftarrow$ node.label
3: **if** (node[node.length()-1] == 0) **then**
4: sibling[sibling.length()-1] $\leftarrow$ 1
5: **else**
6: sibling[sibling.length()-1] $\leftarrow$ 0
7: **end if**
8: ssKey $\leftarrow$ skey(sibling)
9: snode $\leftarrow$ DHT-get(ssKey)
10: **if** ((snode.label.length(() $==$ node.label.length()) && (snode.content.size() + node.content.size() $<$ B)) **then**
11: sum $\leftarrow$ node.content $\cup$ snode.content
12: **if** (node.label[length()-1] == node.label[length()-2]) **then**
13: pnode $\leftarrow$ node
14: dnode $\leftarrow$ snode
15: **else**
16: pnode $\leftarrow$ snode
17: dnode $\leftarrow$ node
18: **end if**
19: pnode.label $\leftarrow$ pnode.label.substring(0, pnode.label.length()-1)
20: pnode.content $\leftarrow$ sum
21: pskey $\leftarrow$ skey(pnode.label)
22: DHT-put(pskey, pnode)
23: dnode.content $\leftarrow$ $\emptyset$
24: dnode.status $\leftarrow$ EXTERNAL
25: dskey $\leftarrow$ skey(dnode.label)
26: DHT-put(dskey, dnode)
27: **end if**
28: **end if**

SPT relies on function `mergeIfNeeded()` (see Alorithm 2 to merge sibling nodes if such an action is required. First, this function checks if the number of remaining records is less than $(\frac{B}{2})$ where B is the maximum capacity of leaf node (line 1). If this condition is satisfied, the function computes the identifier of the sibling node then retrieves its corresponding data from the DHT store (lines 2 – 9). Once data is retrieved, the merging process continues if the following conditions are satisfied : (i) the sibling did not split and (ii) the size of the union set of content is less than B (line 10) If a merging is required, as for split, the content of the sibling with its rightmost bit different from the rightmost bit of the parent is added to the contents of the other. Then both siblings are updated, the one with the same rightmost bit as the parent is updated to become the parent while the other is updated to become an external node (lines 11 – 27)

4.2. SPT-Lookup Primitive

Given `ikey`, this primitive returns the identifier of the node in charge of that indexing key. Algorithm 3 sketches how this primitive proceeds.

Algorithm 3 SPT-Lookup(sid, ikey) : looks up the node in charge of `ikey` located within the sub-tree identified by sid

Require: sid // The identifier of the subtree where to look up
Require: ikey // The indexing key of the summary to locate
Ensure: nid // The identifier of the node in charge of key
1: prefix ← sid
2: inc ← ""
3: notYetFound ← true
4: **while** (notYetFound) **do**
5: previousPrefix ← prefix
6: prefix ← previousPrefix ⊙ inc
7: rest ← key.substring(prefix.length()-1, ikey.length())
8: rlen = rest.length()
9: dhtKey ← skey(prefix)
10: node ← `DHT-get(dhtKey)`
11: **if** (node.status == EXTERNAL) **then**
12: prefix ← previousPrefix
13: mid ← inc.length() ÷ 2
14: inc ← inc.substring(0, mid)
15: **else**
16: **if** ((node.status == INTERNAL) || ((isPrefix(node.label, ikey) == false))) **then**
17: inc ← $\mathcal{SPW}one$(rest, rlen)
18: **else**
19: nid ← node.label
20: notYetFound ← false
21: **end if**
22: **end if**
23: **end while**
24: **return** nid

`SPT-lookup` checks successively the subtree root node and its descendants leaf nodes mapped to storage keys obtained by concatenating to "/" prefixes of `ikey` with the right-

most bit equal to 1. These prefixes of `ikey` are considered in their length order and are constructed incrementely thanks to function $\mathcal{SPW}one$ which determines, at each time, the increment to add to the previous prefix to obtain the next satisfying prefix. In case `SPT-lookup` jumps to an external node after adding some increment, similarly to the traditional binary lookup method, it steps back by reducing the length of the last added increment by half. Once the leaf node in charge of `ikey` is reached (that is a leaf node whose the identifying label is a prefix of `ikey`), this label is returned and the `SPT-lookup` primitive terminates. Note that `SPT-lookup` relies on two basic functions : $\mathcal{SPW}one$ and $isPrefix()$.

$\mathcal{SPW}one(bs, l)$ returns the shortest prefix of bs with at most l bits and its rightmost bit equal to 1 if such a prefix exists ; otherwise 0 (see Equation 2). In this equation, z is a sequence bits 0 while [0|1]* is any sequence of bits.

$$\mathcal{SPW}one(s,l) = \begin{cases} z1, & if\ s = z1[0|1]*\ and\ len(z) < l \\ 0, & if\ s = z[0|1]*\ and\ len(z) \geq l \\ 1, & otherwise \end{cases} \quad (2)$$

Function $isPrefix()$ returns true if the first string is a prefix of the second one.

To illustrate how this works, let consider an SPT where each leaf node is identified by a path of length 5. Suppose we want to locate the leaf node in charge of a summary whose the indexing key is `ikey = "1000000011000001"`. For that, one issues `SPT-lookup("/", ikey)` which causes the following operations on the DHT. (i) `SPT-lookup` performs `DHT-get("/")` and finds that the root is an internal node. One needs to check longer prefixes with the rightmost bit to 1. The set of prefixes of $ikey$ with the rightmost bit equal to 1 is

$$\{"1", "100000001", "100000000011", "1000000011000001"\}$$

Next, `SPT-lookup` performs `DHT-get("/1")` and finds that the identifier of the leaf node currently stored with storage key "/1" is "/1111" (the only label of length 5 mapped to storage key "/1"). Since "1111" is not a valid prefix of `ikey`, `SPT-loopup` considers the next prefix, "100000001", and performs `DHT-get("/100000001")`. The result is an external node. `SPT-lookup` steps back and consider "/10000" then computes the storage key corresponding to it. This gives "/10". `SPT-lookup` performs `DHT-get("/10")` which returns the leaf node identified by "/1000" accordingly to our initial assumptions.

Compared to the traditional linear and binary lookup methods, `SPT-lookup` is somewhat hybrid. It worths to note that the number of `DHT-get()` performed in order to locate the SPT node in charge of `ikey` is less or equal to $n + 2$, where n is the number of bits 1 within `ikey`.

5. SPT Superset Search

Let qs be some keyword-set summary. The superset search primitive (Algorithm 5) aims to determine the set $\mathcal{R}$ of indexed summaries that are descendants of qs (i.e., $\forall r \in \mathcal{R}, r \hookrightarrow qs$).

One key function used by the superset search protocol is `getBranches()`. Given a label `p`, branches of `p` are the set of node identifiers obtained by changing the rightmost bit of prefixes of `p`. For a superset search query associated with the indexing key `ikey`, only branches of the length of prefixes of `ikey` with the rightmost bit equal to 0 are

interesting. Function `getBranches(path, ancestor)` (see Algorithm 4 returns the set of identifiers of interesting branches of `path` descendants of `ancestor`.

Algorithm 4 getBranches(path, ancestor) : returns branch identifiers

Require: ancestor // The identifier of the common ancestor of branches
Require: path // The path that links those branches
Ensure: bstack // The stack of branch identifiers
1: bstack $\leftarrow \emptyset$
2: alen $\leftarrow$ ancestor.lenght()
3: plen $\leftarrow$ path.length() ;
4: **if** (plen $\leq$ alen) **then**
5: **return** bstack ;
6: **end if**
7: lastBit $\leftarrow$ path[plen - 1]
8: j $\leftarrow$ plen ;
9: **while** (j $>$ alen) **do**
10: j $\leftarrow j - 1$
11: **if** (path[j] $==$ 0) **then**
12: **if** (lastBit $==$ 0) **then**
13: bid $\leftarrow$ path.substring(0,j) $\odot$ 1
14: **else**
15: bid $\leftarrow$ path.substring(0,j) $\odot$ 0
16: **end if**
17: bstack.push(bid)
18: **end if**
19: **end while**
20: **return** bstack

To start, a number of variables used by the search protocol are initialized (lines 1 – 4). In particular, `SPT-supersetSearch()` computes the indexing key associated with the query (line 2) and intializes the stack variable `bstack` with the root's label (lines 3 and 4).

The remaining of the algorithm consists mainly on a loop that implements the superset search protocol (lines 4 – 24). At each round the search protocol performs four actions. Firstly, it set `bid`, the identifier of the subtree where to search for this round. This is done simply by taking the node identifier on top of the stack `bstack` (line 6). Secondly, the protocol determines `dhtKey`, the DHT storage key of either the rightmost or the leftmost descendant of node identified by `bid` that can store supersets of qs (lines 7 – 15). Once `dhtKey` is computed, the third action is to access the DHT store and to retrieve the bucket stored within the DHT with `dhtKey`. Upon reception of this bucket, summaries that are superset of qs are added to the set $\mathcal{R}$ of responses. The final action of each round is to determine the set of new pertinent subtrees that need to be explored (line 19) and adds its elements to the current `bstack` (lines 20 – 23)

Once `bstack` becomes empty, $\mathcal{R}$ contains all indexed summaries that are supersets of qs (line 25).

Algorithm 5 SPT-supersetSearch(qs) : returns the set of indexed summaries that contain qs

Require: qs
Ensure: $\mathcal{R}$
1: $\mathcal{R} \leftarrow \emptyset$
2: qskey ← computekey(qs)
3: rootPath ← "/"
4: bstack.push(rootPath)
5: **while** (bstack $\neq \emptyset$) **do**
6: bid ← bstack.pop()
7: blen ← bid.length()
8: **if** (bid[blen -1] == 1) **then**
9: nid ← bid
10: **else**
11: ks ← qskey.substring(blen -1, qs.length)
12: sskey ← bid $\odot$ ks
13: nid ← SPT-lookup(bid, sskey)
14: **end if**
15: dhtKey ← skey(nid)
16: node ← DHT-get(dhtKey)
17: sset ← node.retrieveSupset(qs)
18: $\mathcal{R} \leftarrow \mathcal{R} \cup sset$
19: ibs ← getBranches(node.label, bid)
20: **while** (ibs $\neq \emptyset$) **do**
21: ib ← ibs.pop() ;
22: bstack.push(ib) ;
23: **end while**
24: **end while**
25: **return** $\mathcal{R}$

6. Performance Evaluation

This section presents the performances of a prototype implementation of SPT. We implemented SPT in java and simulated a DHT with an array of storage nodes, each capable to store B records. For this evaluation, we considered a real dataset composed of 4,636,000 abstracts[1] of Wikipedia, which was subsequently treated and subdivided into smaller derived datasets. Table 1 shows the characteristics of the 3 derived datasets used for the evaluation. For all the experiments, documents are summarized by Bloom filters of 1024 bits, using 5 hash functions. We compare the performance for the SPT proposal with the ones of LIGHT [8]. For this comparison, we convert each description into a float number. It worth to note that such a conversion can have an impact on this comparison.

Finally, each document is summarized by a Bloom filter of length 1024 bits, using 5 hash functions.

In the following, we first study the structural performances of the SPT solution. For that, we consider the utilization rate of leaves and the maximal storage capacity of leaves. Secondly, we study the performance of the split. For that, we consider the ratio of data transferred to a another DHT location when a node is split. We also measure the performance of the hybrid lookup procedure. And finally, we evaluate the performance of the

1. http://data.dws.informatik.uni-mannheim.de/dbpedia/2014/en/long_abstracts_en.ttl.bz2

Tableau 1 – Datasets descriptions

Dataset name	Number of words in a summary	number of summary
wiki1	$1 \leq$ nb words < 10	481380
wiki4	$40 \leq$ nb words < 60	618164
wiki5	$60 \leq$ nb words < 80	336373

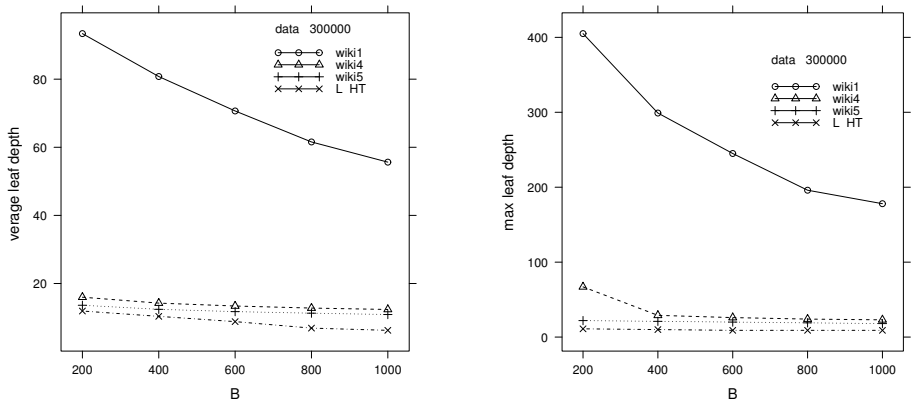


Figure 1 – Structural properties : Mean leaf depht as function of B (a) and Max leaf depth as function of B (b)

superset search protocol. Note that in the experiments varying the size of the dataset, we expand the chosen dataset with documents of same characteristics.

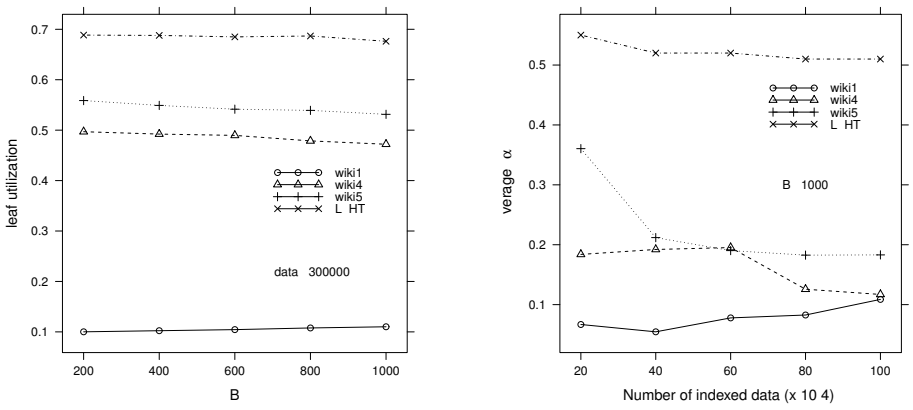
6.1. Structural performance

The following experiments target structural properties of the SPT index, including average leaf depth and leaf utilization. Leaf utilization is defined to be the ratio of the number of records stored in a leaf to the leaf capacity B. We run several tests using different datasets. In each test, we insert a chosen number of records into SPT and we measure the properties.

First we measure the depth of the Summary Prefix tree while varying the size of the indexed data and B. Figures 1a and Figure 1b show that data from wiki1 causes a greater depth than data from wiki4 or wiki5. The reason is that with wiki1, Bloom filters are sparse with very few bits set to 1.

This results in skewed tree. We also remarks that the depth of SPT leaf nodes decreases when the maximal capacity of leaf nodes increase.

In a second step, we evaluated the utilization rate of SPT leaves. Figure 2a shows that the utilization rate from wiki1 is very low ; when indexing documents from wiki 4, it is close to the normal rate that is 0.5 while for data from wiki 5, this rate is higher than 0.5. The explanation for this result is that when a leaf split, its contents must be distributed between the two new leaves. Since the distribution is based on the data (index key), the



(a) (b)

Figure 2 – Leaf utilization as function of B (a), Average α as a function of the index size (b)

bit corresponding to the depth of the leaf determines the location of each data item. So with data from wiki1, the abundance of 0 bits implies that the SPT tree is very leaning to the left with many underused leaves. Then, average leaf utilization is very low. On the other hand, for the data from wiki4 we have an equitable distribution of the data that is justified by the fact that the bit corresponding to the depth of a split node must have the same probability of being 1 or 0. For wiki 5, the reverse scenario of wiki 1 case tends to occur since the words used to describe data from wiki5, thus, Bloom filter contains many 1 bit.

6.2. Index Maintenance Performance

We measure the ratio α, of records moved to a remote peer during a leaf split. For that, we continuously insert data into the index and log the average value of α at each split. During this experiment B is fixed and is equal to 1000. Figure 2b plots the ratio of data moved during our simulation. For SPT, in average, less than 20 percent of data are migrated to a remote node after a leaf split ; also, the higher the number of keywords in each description, the higher the average ratio of data moved to a remote DHT node on each split. Compared to LIGHT, SPT offers better performances, thought one needs to evaluate the impact of data conversion on the LIGHT performances.

6.3. Lookup Performance

To evaluate the lookup primitive, we run three series of experiments, one for each dataset. For each experiment, we first insert a predefined number of records from one dataset within the index, and then run 1000 lookups operations for documents peaked from the same dataset. We record the number of DHT-accesses for each lookup operation. Figure 3a reports the average number of DHT accesses per lookup for each experiment as a function of the index size and the dataset. The number of DHT accesses required to complete an SPT-lookup varies between 2 and 7. Also, the number of DHT access

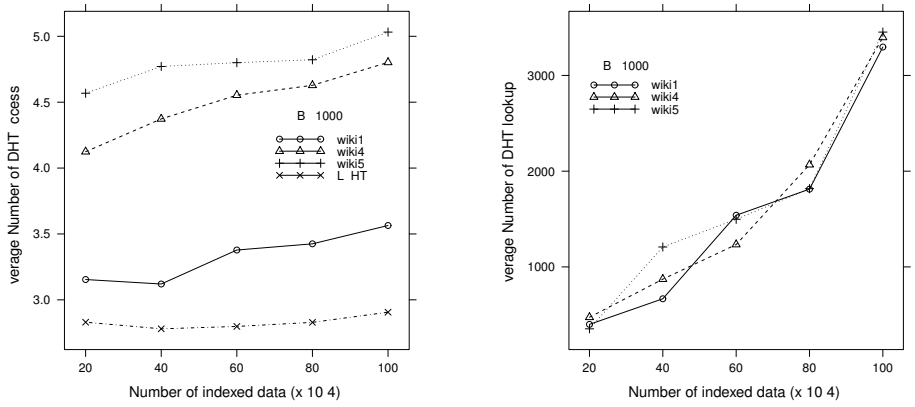


Figure 3 – Number of DHT Access as a function of the index size (a), Number of DHT-Lookup as a function of the index size (b)

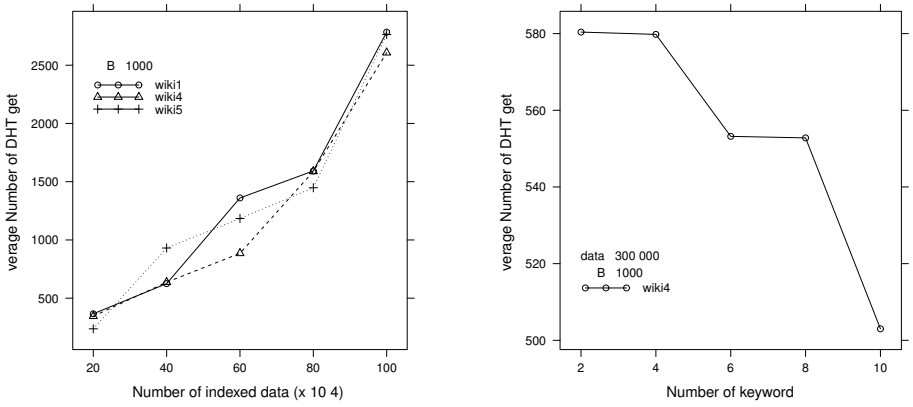
increases, though very smoothly, as the size of the data increases. This figure shows finally that LIGHT performs better than SPT.

6.4. Search Performance

We run experiments with different index sizes while maintaining B equal to 1000. For each experiment, after the insertion of the suitable number of records, we perform a number of superset searches and measure the average number of lookup and of get operations performed on the underlining DHT in order to retrieve all indexed summaries that are descendants of the query summary. It worth to note that the number of DHT-GET performed on behalf of a search request corresponds to the number of SPT leaf nodes whose labels are prefixes of supersets of this request summary and which must be accessed in order to retrieve the set of supersets that satisfy the search request. Note also that one SPT-lookup (cf. Algorithm 3, line 9 and line 10) is required to determine each SPT leaf node responsible of a prefix of a superset of a request summary. From figure 3b and figure 4a, we observe that, for each experiment, the average number of DHT-lookup is less than twice the average number of DHT-get. This suggests that our search protocol is particularly efficient in locating SPT leaf nodes that potentially store supersets of the query summary. We also evaluate the performance of the search according to the number of keywords contained in a query. Figure 4b shows that the number of DHT-get decreases when the number of keywords in a search request increases.

7. Conclusion

We presented SPT, a trie index data structure that can help build a scalable, fault-tolerant and robust over DHT index for keyword-based search. Our solution uses a hybrid lookup procedure that suits the specific of sparse indexing keys. Our extensive experimental evaluation with Wikipedia dataset comprising millions of documents demonstrates the efficiency of our solution. Few data are moved during split operation and the SPT lookup



(a) (b)

Figure 4 – Number of DHT-Get as a function of the index size (a), Number of DHT-Get as a function of the queries sizee (b)

operations are efficient reducing the search cost to the minimum necessary. We are currently working on improving our proposal to compress Bloom filters in order to reduce the storage cost of DHT nodes.

8. Bibliographie

[1] REYNOLDS PATRICK, VAHDAT AMIN, « Efficient peer-to-peer keyword searching », *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*.

[2] WANG MENG-FAN, ZHANG DA-FANG, TIAN XIAO-MEI, BI XIA-AN, ZENG BIN, « Multi-keyword search over P2P based on Counting Bloom Filter », *2nd International Conference on Networking and Information Technology IPCSIT, 2009*.

[3] CHEN, HANHUA, JIN, HAI, CHEN, LEI, LIU, YUNHAO, NI, LIONEL M., « Optimizing Bloom Filter Settings in Peer-to-Peer Multikeyword Searching », *IEEE Transactions on Knowledge and Data Engineering*, vol. 24, n° 4, 2012.

[4] CHEN, HANHUA, JIN, HAI, WANG, JILIANG, CHEN, LEI, LIU, YUNHAO, NI, LIONEL M., « Efficient Multi-keyword Search over P2P Web », *17th International World Wide Web Conference*, 2008.

[5] LINTAO LIU, KYUNG DONG RYU, KANG-WON LEE, « Supporting efficient keyword-based file search in peer-to-peer file sharing systems », *Global Telecommunications Conference, GLOBECOM '04. IEEE*, vol. 2, 2004.

[6] JOUNG YUH-JZER, FANG CHIEN-TSE, YANG, LI-WEI, « Keyword Search in DHT-Based Peer-to-Peer Networks », *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems, 2005*.

[7] SRIRAM RAMABHADRAN, SYLVIA RATNASAMY, JOSEPH M. HELLERSTEIN, SCOTT SHENKER, « Brief announcement : Prefix hash tree », *23rd ACM Symposium on Principles of Distributed Computing, 2004*.

[8] YUZHE TANG, SHUIGENG ZHOU, JIANLIANG XU, « Light : A query-efficient yet lowmaintenance indexing scheme over dhts », *IEEE Trans. on Knowl. and Data Eng.*, vol. 22, n° 59-75,

2010.

[9] NICOLAS HIDALGO, LUCIANA ARANTES, PIERRE SENS, XAVIER BONNAIRE, « A tabu based cache to improve latency and load balancing on prefix trees », *17th IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2011*.

[10] RUDYAR CORTÉS, XAVIER BONNAIRE, OLIVIER MARIN, LUCIANA ARANTES, PIERRE SENS, « Geotrie : A scalable architecture for location-temporal range queries over massive geotagged data sets », *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA), 2016*.

[11] TIGELAAR ALMER S., HIEMSTRA DJOERD, TRIESCHNIGG DOLF, « Peer-to-Peer Information Retrieval : An Overview », *ACM Trans. Inf. Syst.*, vol. 30, n° 2, 2012.

[12] JIN, XING, YIU, WAI-PUN, CHAN, S.-H. GARY, « Supporting Multiple-Keyword Search in A Hybrid Structured Peer-to-Peer Network », *EEE International Conference on Communications (ICC), 2006*.

[13] CHANGXI ZHENG, GUOBIN SHEN, SHIPENG LI, SCOTT SHENKER, « Distributed segment tree : Support of range query and cover query over dht » *Electronic publications of the 5th International Workshop on Peer-to-Peer Systems (IPTPS), 2006*.

[14] LI FAN, PEI CAO, JUSSARA ALMEIDA, ANDREI Z. BRODER, « Summary cache : A scalable wide-area web cache sharing protocol », *IEEE/ACM Trans. Netw.*, vol. 8, n° 3, 2000.