Taming Hot-Spots in DHT Inverted Indexes

Nuno Lopes[*]
CCTC-Department of Informatics
University of Minho
Braga, Portugal
nuno.lopes@di.uminho.pt

Carlos Baquero
CCTC-Department of Informatics
University of Minho
Braga, Portugal
cbm@di.uminho.pt

ABSTRACT

DHT systems are structured overlay networks capable of using P2P resources as a scalable platform for very large data storage applications. However, their efficiency expects a level of uniformity in the association of data to index keys that is often not present in inverted indexes. Index data tends to follow non-uniform distributions, often power law distributions, creating intense local storage hotspots and network bottlenecks on specific hosts. Current techniques like caching cannot, alone, cope with this issue.

We propose a new distributed data structure based on a decentralized balanced tree to balance storage data and network load more uniformly across all hosts. The approach is stackable with standard DHTs and ensures that the DHT storage subsystem receives an uniform load by assigning fixed sized, or low variance, blocks.

1. INTRODUCTION

Distributed Hash Tables (DHTs) are structured overlay networks capable of efficiently storing and locating objects from a given key. Systems like Chord, Pastry and CAN [20, 18, 16] allow scalability in the number of hosts, requiring only logarithmic communication steps and routing state. A hash function is used to uniformly distribute keys to hosts so that key load is balanced.

This perfect distribution has two intrinsic assumptions: Keys are uniformly accessed, both in storage and retrieval; The size of the tuples $\langle key, object\rangle$ depict a low variance. However, these assumptions are often not possible. This is the case when building term-partitioned inverted indexes over DHTs [13, 17, 22], where words are mapped to the locations of the documents where they occur.

Hot spots created by data or query asymmetries will occur due to the power-law distribution of text keyword frequency [25]. When a single key is accessed very often (e.g. "Katrina"), a network bottleneck appears on the host storing that key. This situation known as "query flash crowd" can be minimized with caching schemes [19, 14]. On the other hand, storage hotspots occur when very large objects of skewed size are stored on individual DHT keys (e.g. the occurrence set for the word "the"). Although storage is often not a critical resource, due to the current trend on secondary storage capacity, storing such large objects creates an additional network bottleneck on the hosts mapping these keys. These network bottlenecks limit the scalability of term-partitioned indexes [1] and cannot be eliminated by caching, as caching is effective only when reading data and not when new data is being inserted into the system. Furthermore, solutions that dynamically redistribute keys across hosts [10, 15] are also unable to eliminate the storage hotspots because the storage unbalance is due to a single key containing a very large object.

In this paper we propose a solution for load balancing DHTs when storing (decomposable) objects with high size variance. We developed a new DEcentralized Balanced tree (DEB) tree algorithm capable of converting a very large object into multiple bounded size blocks suited for being stored and searched as objects over DHTs. We used the DEB algorithm to build a textual inverted index, allowing multiple keys retrieval. The system evaluation shows expressive improvements in the storage and network load distribution, for both index population operations and data retrieval.

Our paper is organized as follows: Section 2 shows an overview of the system interfaces, Section 3 describes the DEB tree algorithm and Section 4 the text indexing system. Section 5 shows our evaluation results, and Section 6 presents the related work and is followed by the Conclusions.

2. SYSTEM OVERVIEW

The system provides an inverted index interface to user applications and stores the index on a structured P2P overlay that is implemented by a DHT algorithm. The system architecture, depicted in Figure 1, is composed by the index layer that presents an inverted index interface to client applications, the tree management layer that implements our distributed balanced tree algorithm and the routing layer (the DHT algorithm) responsible for routing messages between hosts. The index layer receives requests from client applications and converts them into tree based operations to be executed by the tree layer. In turn, the tree layer uses the routing layer to locate the tree block that should process the operation. Tree blocks are stored on the P2P sub-system.

2.1 Index Model

[*] Supported by a Ph.D. Scholarship from FCT - Foundation of Science and Technology, the Portuguese Research Agency.

ACM SIGIR Workshop on Large Scale Distributed Systems for Information Retrieval '07, Amsterdam, The Netherlands.
.

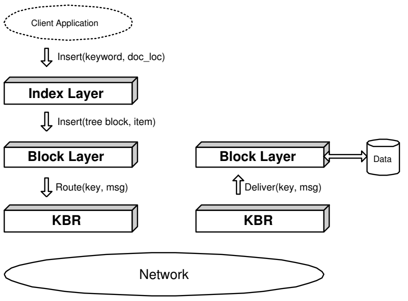

Figure 1: The system is built with a base DHT overlay network for managing hosts membership in a scalable way. Each host of the system contains three components: a key-based routing (KBR) layer, the block storage module and the client index interface.

- INSERT (*keyword*, *doc_loc*): *status*
- REMOVE (*keyword*, *doc_loc*): *status*
- SEARCH ($keyword_{set}$): doc_loc_{set}

(a) Index Layer Interface

- BLOCK-ITEM-INSERT(*key*,*item*):*status*
- BLOCK-ITEM-REMOVE(*key*,*item*):*status*
- BLOCK-GET(*key*):*block*

(b) Block Layer Interface

Figure 2: Layered Interface.

A textual inverted index stores relations between text words (the vocabulary) and sets of document locations (the occurrences) [2] in the form:

$$keyword \mapsto \{document\ location\}_{SET}.$$

Since a single keyword can occur on multiple documents, we store a set of document locations for each keyword. Document locations are just single opaque objects capable of locating a document over the system. The pair $\langle host\_address, docId_{local}\rangle$ is an example of a simple location scheme. Other location schemes could be used, like an URL link or the document content hash value, provided the retrieval of the document is possible from the location value [13, 21].

The index is made accessible to system peers through the interface on Figure 2(a). User applications, in any given node, contact the local index library through this interface. The index INSERT operation adds a new relation between a keyword and a document. Likewise, REMOVE cancels an association. The index search operation retrieves the list of documents associated with a keyword or a set of keywords.

We only considered the *and* Boolean operator for multiple keyword queries, although the remaining Boolean operators could also be implemented [2].

2.2 DHT Interface

The DEB Tree implementation uses a custom DHT interface that substitutes the typical GET/PUT interface. These operations, shown in Figure 2(b) allow a fine grain manipulation of the data object associated to a given key. Here the object has a Set structure and the operations allow the insertion and removal of individual items. Otherwise, if the usual GET/PUT operations were used, latency would double and consistency problems could arise due to lack of atomicity in GET,PUT sequences [4].

The actual custom DHT interface is slightly richer, in order to support other needed operations which are best performed on the node that hosts the block. Their implementation does not present additional difficulties once a Key Based Routing interface is available, ROUTE($key, message$), which is the case for all DHT implementations [6].

3. DEB TREE LAYER

We will now describe our DEB tree implementation. This tree algorithm was based on the B^+-tree design [5] and shares the high-availability requirements present on B-link trees [9]. However, unlike the B-link tree algorithm which was designed for a cluster based architecture with global system view and centralized environment, our algorithm was designed for being deployed on wide-area systems requiring neither global knowledge nor centralized entities.

The DEB-tree algorithm supports a mapping interface for $\langle key, value\rangle$ tuples, just like the B^+-tree, storing document locations (the key) and opaque payloads (the value). In this paper only document locations were used, hence the absence of a value payload on the block insert operation (see Figure 2(b)). Each DEB tree instance stores the document location set for a particular keyword. Therefore, each index keyword will have an unique DEB tree.

3.1 Tree Structure

The tree structure, just like in the B^+-tree design, is composed by a root block and child blocks, the last level of blocks (further away from the root) are called leaf blocks. Leaf blocks store data items and are all at the same tree level (any leaf is accessed from the root block with the same number of block hops). Internal blocks serve exclusively for locating leaf blocks and do not contain any data, instead they contain child block keys.

All blocks contain a parent's field with the key to the upper level block. To improve availability, each block also stores the key to the next sibling block, following the B-link design.

The number of items in any block is bounded by the tree's degree t which defines the minimum $(t-1)$ and maximum $(2t-1)$ number of elements allowed inside a block [5]. For internal blocks the degree influences the number of child block keys it contains. For leaf blocks it influences the number of document locations the block stores.

In addition to the previous fields, each block contains the minimum and maximum limits, representing the interval of data, in terms of key ids, the block is responsible for. The root block has the whole key interval, covering all data. The sum of intervals at each tree level also covers the whole interval, and in a given level all intervals are disjoint.

3.2 Block Identification Scheme

Each tree block is identified by an unique key and stored on the DHT using the hash value of it's key. Since we store all tree blocks under the same name space, the DHT hash domain, we must ensure all blocks will have an unique identification.

Decentralized generation of (probabilistic) unique block ids is made by hashing a globally unique triplet that consists of $\langle keyword, level, minlimit \rangle$. Notice that each stored *keyword* (See index interface) gives rise to a distinct tree, and that all blocks in that tree will depict a distinct level number and minimum key limit on their local key range.

3.3 Tree Layer Algorithms

When client applications, in a node, request operations at the index interface the respective algorithms are executed and, possibly, tree management algorithms are triggered. These last algorithms are responsible for splitting or merging blocks.

Index interface operations are decomposed into one or more DHT block operations. These operations are issued from the node hosting the client application. On the contrary, tree management operations are triggered and issued in the node hosting the block that needs splitting or merging.

All these algorithms are made tolerant to concurrency issues on what concerns structural integrity of the tree and the stored data. To achieve this, some item insert operations may be delayed. Notice that inserts can even be made to timeout and be repeated, since the insert operation is idempotent.

An index SEARCH operation that covers data modified by a concurrent INSERT may, or may not, see the effect of the insert. However, once an insert completes on a client host, subsequent searches in that host or in any other host will see the insertion if the data is covered by the search.

This is achieved even in the presence of caching, since the algorithms only cache internal blocks, and these only contain references to other blocks. Stale information only enacts a performance penalty.

3.4 Data Resilience and Structural Repair

Basic resilience to host failures and churn should be provided by the underlying DHT algorithm. In particular it is desirable to use DHT solutions, like [14], that replicate more intensively blocks that are subject to more activity.

However, even in the event of a fault that is not masked by the DHT layer it is possible to recover the structural integrity of the tree by making use of the redundancy in the structure. The loss of an internal block does not remove relevant data from the tree. Faults at the leafs, however, lead to lost association between keys and locations. The lost data in this case can only be recovered by re-announcing at the clients.

4. INVERTED INDEX OPERATIONS

The index operations amount to inserting references and searching for keywords. These operations are available at the client's host and issue multiple block requests through the DHT to accomplish the initial index operation.

4.1 Document Insertion

For indexing a document into the system, peer clients use the INSERT ($keyword, doc\_location$) function, which adds a document location to a keyword occurrence set. Since the occurrence set is stored on a DEB tree instance, one tree per keyword, this is to say the document location will be inserted into the corresponding DEB tree. The client must call the INSERT function for every $\langle keyword, doc\_location \rangle$ pair it wishes to index.

```
procedure insert (keyword, doc-location):
blk-key ← getRootBlockKey (word)
route (blk-key,⟨ insert, doc-location⟩)
answer ← wait for returning message
while answer ≠ 'ack':
        blk-key ← get forward block from answer
        route (blk-key,⟨insert, item⟩)
        answer ← wait for returning message
end while
```

(a) Client side index insertion

```
procedure insert-block (block, item):
if leaf(block):
    insert(block,item)
    ret message ⟨ack⟩
else:
    ret message ⟨forward, successor(block, item)⟩
endif
```

(b) Block side index insertion

Figure 3: Simple pseudo-code for the index insertion procedure.

Tree insertion is made first by locating the block responsible for storing the item and then by inserting it on the block's data. If the tree only contains a single block, the root block, then the operation finishes after accessing this block. For bigger trees, the client starts at the root block and follows child block references until reaching the correct leaf block. The operation terminates after receiving the acknowledgment of the insertion from the leaf. Figure 3(a) shows simple pseudo-code for the insertion operation on the client index side. Removing an index occurrence works in the same way as for the insertion case.

Inserting a document location on a keyword occurrence set requires the insertion of an item on the keyword set tree. Inserting an item on a B-Tree with I items uses $O(\log I)$ block accesses, which corresponds to the tree height [5]. Each block access is made by the client host using the ROUTE function supplied by the DHT algorithm, which in turn uses $O(f(N))$ messages to locate the host for a key, having f as the lookup cost function on the DHT (typically the logarithmic function). The number of messages used to insert an index occurrence on a system will grow $O(\log I \cdot f(N))$ for N hosts. Since some DHT implementations only require $O(1)$ steps to locate a key [8, 14], this results in $O(\log I)$ complexity. Common DHT algorithms provide a logarithmic cost and therefore will use $O(\log I \cdot \log N)$ messages.

If a client's local cache is used for caching top level tree blocks, the client can access the target leaf block directly for insertion, reducing the complexity even further to $O(1 \cdot f(N))$ for inserting a single reference onto the index.

4.2 Multiple Keyword Search

Queries in this index system follow a multiple keyword intersection model, using the *and* Boolean Query operator for returning the set of document locations that are common to all the query keywords. To perform this intersection, the client would need to fetch all the occurrence sets and then perform a local intersection on the fetched data to determine the final result set.

This simple solution is clearly not optimal. Fetching a complete large occurrence set uses network bandwidth to retrieve data that may not be necessary to effectively answer the query. Remember that each keyword occurrence set is stored under a different DEB tree instance.

We opted for an incremental intersection evaluation that makes a parallel breadth-first traversal of all the trees simultaneously. The use of an incremental evaluation enables the use of two optimization techniques to considerably reduce the query network bandwidth: early-pruning and term reordering. The early-pruning heuristic was inspired by the adaptive set intersection algorithm suggested by Li et al. [11] to minimize data exchange when evaluating set intersections. The heuristic prunes tree sub-branches according to the rule that intersecting an empty set with any set will always be empty. By selecting the branches of large trees to visit according to items already found on smaller trees, and pruning the remaining branches, the heuristic reduces the number of visited blocks without affecting the operation's correction.

Term re-ordering is a database optimization that consists in accessing the intersection sets in order from the smallest to the largest so that the amount of exchanged data is reduced to the minimum. The re-ordering was implemented by accessing first the root blocks of smaller trees. The size of each keyword set was locally determined by the tree's height, which is available at the root blocks.

A single keyword query retrieval requires a tree vertical traversal until reaching the leaf level and then a linear leaf block traversal. Just like in the insertion case, a vertical traversal uses $O(\log I)$ block accesses and the leaf level grows with the number of items in the set $O(I)$, so that the total cost of the operation will be $O(I)$ on the number of items. Since each block access requires $O(f(N))$ host messages, the total number of messages on the system will be $O(f(N) \cdot I))$. Assuming an $O(1)$ hop DHT is used, a single keyword retrieval requires $O(I)$ messages.

The number of stored items I follows a power-law distribution on the number of documents. The value of I depends therefore on the popularity of the keyword. For a few very popular keywords (in storage frequency), I grows linearly with the number of documents. For the remaining keywords it tends to be constant towards the number of documents.

The previous complexity limit assumes that a multi-keyword query uses multiple independent single keyword retrievals. The search optimizations described previously reduce the total number of retrieved items significantly, bringing the query cost, in terms of number of retrieved items, closer to the number of common items instead of the total number of stored items for all keywords in the query.

5. SYSTEM EVALUATION

We will now evaluate the DEB tree index on a textual document collection. Document reference insertion and keyword search will be tested. The evaluation focuses on the load balance properties of the solution when compared to the equivalent linear DHT mapping, considering both storage and network resources.

5.1 Setup

Our DEB tree implementation was deployed on a custom made discrete event simulator implementing a simplified SSF framework written in Python. The communication between hosts and DHT message routing, were simulated. Although our experiments ran over this network simulated environment, the algorithm implementation is made of actual code and could be placed on top of a real DHT system for deployment. In this article, we opted for the simulation model to test the algorithm under a controlled environment with a larger number of hosts.

5.2 Index Insertion

The simulation of the insertion procedure consisted in 1000 hosts concurrently inserting word references, in documents, into the distributed index. Each host was given 10 unique documents to insert. Each document was a newspaper article, with an average size of 3Kb. As expected, this dataset depicts a Zipf distribution, with a few high rank words present in most of the documents.

We evaluated the algorithm performance by varying the tree's block size value. We used a very large block size (shown as $+\infty$) to represent the case of a direct mapping of the index on the DHT [13, 7, 17]. This infinite block will never be full and consequently never split, creating exclusively single root block trees. The other sizes represent the block maximum size for each simulation.

We will now look at the effective load each host received. We assumed a perfect mapping between blocks and hosts. First, we calculated the hash value of the block's key. Then, we assigned a host to the block giving it's hash value. Each host received an equal size share of the hash domain. This allocation overcomes the absence of an actual DHT substrate, but the incurred simplification matches the behavior of an actual DHT with a reasonable number of hosts. It is asymptotically correct.

Figure 4 shows how the storage load distributes, with different block sizes. The infinite block size case shows the less uniform distribution. On the other side, the smaller block sizes show an almost perfect load distribution. However, very small sizes tend to over-split trees, creating excessive internal blocks and increasing the total load on the system. This is the case of the tree with block size 4. A block size of 32 shows an adequate trade-off, and is appropriate to ensure a mostly uniform storage load distribution from an highly skewed dataset.

It is also relevant to analyze how the network load is distributed during insertions. We can expect that root blocks of popular words will have a high demand. The algorithm uses block caches to control this demand.

Figure 5 shows that without caching, smaller blocks have a negative impact as they lead to larger trees and more split operations, increasing the overall message load. This situation would not be adequate.

The same figure shows the results of the same insertion procedure with client cache enabled on hosts. As expected, the variation between the minimum and maximum loaded hosts has decreased significantly for any block size. The

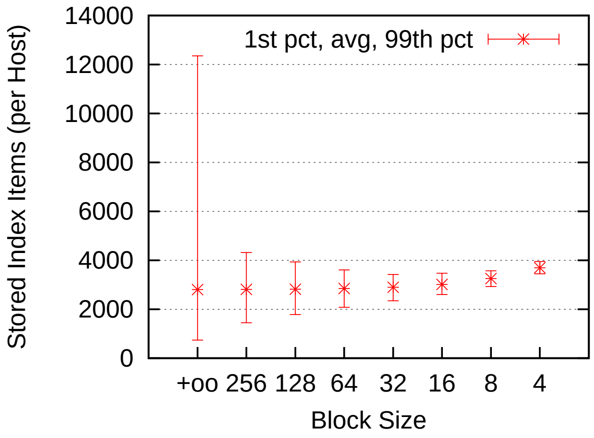

Figure 4: Storage load when inserting. Showing averages, 1^{st} and 99^{th} percentiles.

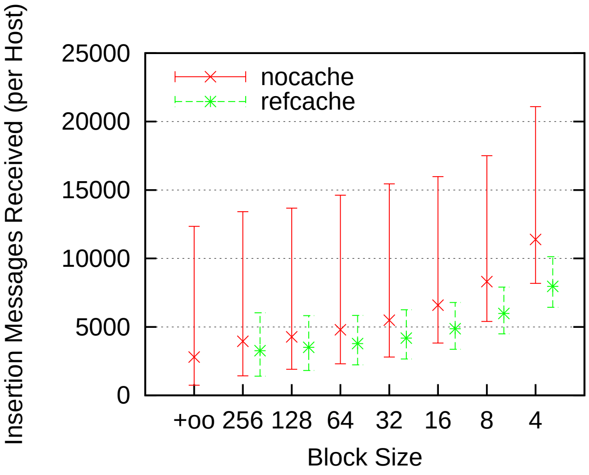

Figure 5: Message load when inserting. Showing averages, 1^{st} and 99^{th} percentiles.

Block size	+oo	256	128	64
No cache	2799	3947	4270	4793
Cache	2799	3264	3503	3784
Block size	32	16	8	4
No cache	5487	6588	8312	11384
Cache	4183	4864	5972	7955

Figure 6: Average Insertion Messages Received per Host

simulation with the very large block size, containing only single root block trees, is identical to the previous simulation without cache because cache only acts on internal blocks. As block sizes get smaller, caching reduces the extra load caused by accessing the top level blocks on larger trees. As a consequence the network load is better distributed across hosts. A block size of 32 would also be an adequate choice, on what concerns data insertion. Table 6 summarizes the average number of insertion messages received per host for different block sizes, with and without cache, found in Figure 5.

5.3 Index Searching

The index searching procedure starts from a fully loaded index. A single host processes a list of multiword search queries. We generated 20000 queries with a multiple keyword distribution from the original text collection. Search keywords follow a Zipf distribution but the rank order was randomized, so that popular keywords are distinct in searches and insertions. The Zipf distribution for generating individual search terms does not consider the correlation between keywords found in real multi-term query traces. We plan as future work to run the same evaluation of the algorithm using a corpus with real query logs.

We measured the index search load by counting the number of index items replied to the caller. Index items correspond to document locations when accessing leaf blocks and correspond to children block limits when accessing internal blocks. We assumed that document locations (and block limits) have a constant or small variation size in bytes.

We will first show the impact of our query optimizations on the system bandwidth. Figure 7 shows the cumulative distribution function (CDF) of the number of block requests (messages) received at hosts, according to the query optimizations used for a block size of 32 items. The worst result appears on the basic incremental method (label `inc`) , which traverses all trees in a breadth-first order. It is followed by the early-pruning method (label `early`) which improves the basic incremental method by stopping the retrieval of further blocks that cannot contribute to the final result set. We improve further by adding a keyword term reordering (label `sort-loc`) that starts by accessing smaller trees first and leaving larger trees to the end. This term reordering works in conjunction with early-pruning to interrupt block retrieval as soon as the final result set can be computed.

The term reordering method was originally developed for local knowledge, so we also simulated a variation (label `sort-glob`) that supplied the client with the global index keyword frequency. This experiment allowed us to determine the maximum possible gain from using this heuristic, although it cannot be used in real systems.

We implemented the reference cache procedure over the local term re-ordering heuristic (label `sort-loc-cache`). When comparing it to the same query method without cache in Fig. 7 (label `sort-loc`), one observes that although cache reduced the overall load, it was only marginally. This performance can be explained by noticing that cache was operating only on internal blocks, having no effect on leaf accesses. Since leafs are not cached, the hosts storing leaf block data for popular keywords are overloaded with requests and hence the high number of messages received at some hosts. A query "flash crowd" on leaf blocks could be handled directly by the DHT layer [14].

Having established that the best usable heuristic is the one depicted by `sort-loc-cache`, we now proceed to analyze the impact of different block sizes. Figure 8 shows, in log scale, for various blocks sizes, the number of items that are sent by hosts in response to the generated query load.

Although smaller block sizes will lead to more messages, on what concerns the number of items sent (in a sense, the bandwidth) smaller blocks are better. Because the block size is smaller, each block request causes a smaller impact on the network load of each host (the quantity of data items

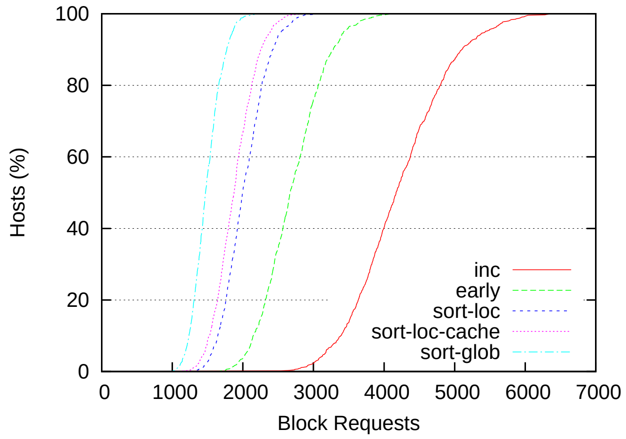

Figure 7: A cumulative distribution function (CDF) of the number of block requests (messages received) on hosts for the different searching methods with a block size of 32 items.

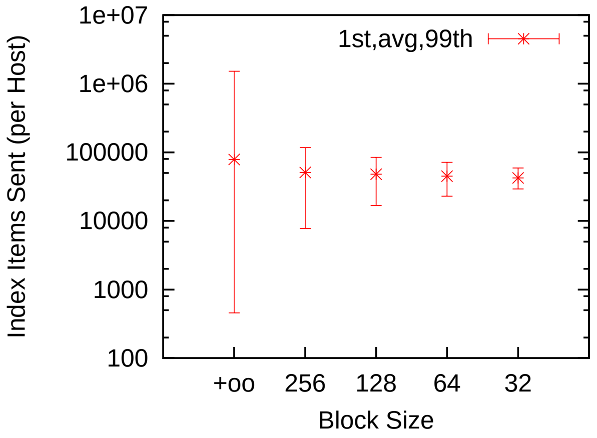

Figure 8: The network load distribution (the 1st percentile, average and 99th percentile on the quantity of index items replied to the client) on hosts. The vertical axis is in logarithmic scale.

transmitted back to the client). This figure also shows that the DEB Tree algorithm uses less network bandwidth on average and with a more uniform distribution across hosts, when executing the same query data set when compared to the direct DHT mapping.

6. RELATED WORK

The related work is separated in two major groups: keyword based and range-query searching algorithms. On keyword search systems the distributed index is either local to each host (partition-by-document) or shared among all hosts (partition-by-keyword) [11]. Overcite [21], a P2P implementation of the Citeseer system with keyword searching, uses a partition-by-document design for their index, requiring clients to search simultaneously on all hosts (or clusters) where each one maintains a local index. Our algorithm assumes a partition-by-keyword design where clients contact the host (or hosts) responsible for a specific keyword on the global index.

Previous work on partition-by-keyword indexing stores the index directly on the DHT and does not handle the storage hotspot problem we have identified [13, 7, 17]. Tang and Dwarkadas [22] proposed a constant factor balancing to deal with the storage hotspot. Our algorithm adapts dynamically to the object size, ensuring always an uniform storage distribution despite the object size variation. All the previous systems can use our algorithm as a middle layer for storing large posting lists as smaller bounded size pieces.

Range-query based systems create structures that enable the search by range limits instead of discrete values. Mercury uses multiple independent rings of peers (based on chord) to support multiple attribute range queries [3]. Brushwood creates a distributed tree using the Skip Graph routing algorithm to store data with locality properties [23]. The DEB-tree algorithm can be used with any DHT algorithm through the basic routing interface and just like the previous, it can also support range-queries [12].

An additional class of algorithms was designed to run over generic DHT systems. Those algorithms offer range-query functionality by constructing tree based structures. Chawathe et al. proposed the use of a Prefix Hash Tree (PHT) for for storing (x, y) coordinates of wireless access points [4]. Zheng et al. presented a Distributed Segment Tree (DST) algorithm with similar functionality but using a static load balancing technique [24]. The Deb-tree algorithm shares with the previous systems it's tree based structure over a generic DHT. It differs, however, by using a structure that adapts dynamically to data, resulting in a perfectly balanced tree.

7. CONCLUSIONS

Although research in DHT systems is still recent, these systems have already proven their important role in the support of scalable distributed data store solutions. However, their promise of a fair load distribution is only met when we just consider the allocation of keys to host nodes. If a key is often accessed or if it holds a huge data object, the associated host no longer receives a fit load. This is often the case in real datasets and in particular when constructing indexes. The world is more often governed by power-laws than by uniform distributions.

In this article we bring attention to this issue and present a system that adapts classical and proven techniques, Balanced Trees, to a demanding distributed setting where the technique can be used to solve the aforementioned problem. Unlike other solutions, we do not redesign from scratch, but instead use a simple adaptation over off-the-shelf DHTs and hopefully benefit from the large amount of research that focused on efficient DHT designs.

We evaluated our algorithm on a concurrent simulated environment with a textual reverse index, a highly skewed dataset, to illustrate it's load balancing properties on both storage and network resources.

The results show that the algorithm is capable of balancing storage load perfectly and reducing the network load variation by half when compared to a direct DHT use when inserting data into the index.

Querying the index also revealed a more uniform network load distribution, reducing the standard deviation from three orders of magnitude to one, when comparing our query optimized algorithm to the direct DHT case.

Acknowledgments

The authors wish to acknowledge Sylvia Ratnasamy and Rodrigo Rodrigues from the Eurosys Doctoral Symposium and the anonymous reviewers for their comments.

8. REFERENCES

[1] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed web retrieval. In *IEEE 23rd International Conference on Data Engineering*, April 2007. Invited Speaker.

[2] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* ACM Press, NewYork, 1999.

[3] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proceedings of SIGCOMM 2004*, August 2004.

[4] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, J. Hellerstein, and S. Shenker. A case study in building layered dht applications. In *Proceedings of the ACM SIGCOMM'05 Conference*, pages 97 – 108, 2005.

[5] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 1989.

[6] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common api for structured peer-to-peer overlays. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, USA, February 2003.

[7] O. Gnawali. A keyword-set search system for peer-to-peer networks. Master's thesis, Massachusetts Institute of Technology, May 2002.

[8] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*, pages 113–126, March 2004.

[9] T. Johnson and P. Krishna. Lazy updates for distributed data structures. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, 1993.

[10] D. Karger and M. Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, USA, February 2003.

[11] J. Li, B. Loo, J. Hellerstein, M. Kaashoek, D. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, USA, February 2003.

[12] N. Lopes and C. Baquero. Implementing range queries with a decentralized balanced tree over dhts. In *Proceedings of 1st International Conference on Network-Based Information Systems (NBiS2007)*, 2007. To appear.

[13] Overnet website. http://www.overnet.com/.

[14] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*, pages 99–112, March 2004.

[15] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems. In *Procs of the 2nd Intl. Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, USA, February 2003.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of the ACM SIGCOMM'01 Conference*, pages 161–172, 2001.

[17] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference*, Brazil, 2003.

[18] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms*, Germany, 2001.

[19] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. In *Procs of the 1st Intl. Workshop on Peer-to-Peer Systems (IPTPS'02)*, Cambridge, MA, USA, March 2002.

[20] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM'01 Conference*, pages 149–160, 2001.

[21] J. Stribling, J. Li, I. G. Councill, M. F. Kaashoek, and R. Morris. Overcite: A distributed, cooperative citeseer. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)*, 2006.

[22] C. Tang and S. Dwarkadas. Hybrid global-local indexing for efficient peer-to-peer information retrieval. In *Proceedings of First Symposium on Networked Systems Design and Implementation*, San Francisco, USA, March 2004.

[23] C. Zhang, A. Krishnamurthy, and R. Wang. Brushwood: Distributed trees in peer-to-peer systems. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS'05)*, New York, USA, February 2005.

[24] C. Zheng, G. Shen, S. Li, and S. Shenker. Distributed segment tree: Support of range query and cover query over dht. In *Electronic publications of the 5th International Workshop on Peer-to-Peer Systems (IPTPS'06)*, California, USA, February 2006.

[25] G. Zipf. *Human Behaviour and the Principle of Least Effort.* Addison-Wesley, 1949.