Search Optimizations in Structured Peer–to–peer Systems

Nuno Lopes* and Carlos Baquero*
nuno.lopes@di.uminho.pt, cbm@di.uminho.pt

*DI/CCTC
Universidade do Minho
Braga, Portugal

***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 distributed data structure based on a decentralized balanced tree to balance storage data and network load more uniformly across hosts. The results show that the data structure is capable of balancing resources, in particular when performing multiple keyword searches.

***Index Terms*—E.1.b Distributed Data Structures, D.4.7.b Distributed Systems (Distributed Hash Tables), H.3.3.a Clustering, H.3.3.h Search Process**

I. Introduction

Peer–to–peer systems have received attention due to their dynamic adaptation to host failures, decentralized architecture and amount of distributed resources (storage and network bandwidth) that are available for distributed applications. One interesting application for such large amount of resources is cooperative searching where each host participates with information (documents) to be searched and with resources to support such (keyword based) search.

When implementing search with the help of an index, two approaches are available to distribute a very large index across a peer–to–peer system: partition-by-document and partition-by-term [1]. The partition-by-document scheme groups documents into multiple independent partitions that contain a local index for their own documents. The partition-by-term scheme creates a single global index where each partition stores the occurrence set (document references) for a keyword (or group of keywords). The partition-by-document scheme is very efficient at building the index, which can be fully parallelized, but queries must be sent to all partitions. On the other hand, the partition-by-term is more costly when building the index but more efficient in resource usage when processing queries [2]. Moffat et al. showed that partition-by-term is more efficient in (query) resource usage when compared to the partition-by-document [1].

Distributed Hash Tables (DHTs) are a particular type of peer–to–peer system that are capable of efficiently storing and locating objects from a given key. Systems like Chord, Pastry, CAN [3], [4], [5] and others 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. Furthermore, DHTs assume that peers are homogeneous.

Several systems build term-partitioned inverted indexes over DHTs [6], [7], [8], where keywords are mapped to the locations of the documents where they occur. This mapping makes use of the DHT uniform hash function to locate the peer responsible for storing the document locations, given a DHT key that corresponds to the index keyword. Although the index model maps perfectly into the DHT interface, the DHT model has two intrinsic assumptions: keys are uniformly accessed, both in storage and retrieval; and the size of the tuples $\langle key, object \rangle$ depict a low variance. These assumptions are often not possible when indexing textual data.

Hot spots created by data or query asymmetries will occur due to the power-law distribution of text keyword frequency. 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 [9]. 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 [2] and cannot be totally 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], [11] 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 present 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 DHT values. We used the DEB algorithm to

build a textual inverted index, allowing multiple keys retrieval. The system evaluation shows a reduction of three orders of magnitude in the network load distribution standard deviation, when considering query optimizations.

II. Related Work

Several systems have proposed the building of a textual inverted index on top of a DHT [6], [7], [8], [12]. Among the previous systems, only Overcite [12], a peer–to–peer implementation of the Citeseer system with keyword searching, uses a partition-by-document design for the search functionality. Both the DEB-tree system and the remaining systems use the partition-by-term scheme over a DHT to store the index.

The systems that store the index directly on top of a DHT do not handle the storage hot-spot problem we have identified [6], [7]. KSS [6] pre-computed the set intersection of all keywords up to some combination length in order to optimize multiple word queries and stored those set intersections on the DHT. Reynolds and Vahdat [7] used a bloom filter to compare multiple keyword sets and reduce the amount of data transmitted between hosts. Both cases try to avoid query network overloading using techniques that reduce the amount of information communicated for answering queries. However, none of them addresses the unbalanced storage caused by inserting very large occurrence sets on the DHT.

Tang and Dwarkadas [8] also proposed to store the index directly over the DHT, however they dealt with the storage hot-spot by using a constant factor balancing, distributing occurrences of the same index keyword through a fixed interval of DKEYs. This constant factor distribution does not take into account the final object size forcing clients to access all DKEYs inside the interval to manipulate data for the index keyword.

The use of the DEB-tree algorithm creates an additional layer that automatically balances the storage load and consequentially the network bandwidth used for creating the index over the DHT hosts.

III. 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 (block) 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.

A. Index Model

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

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

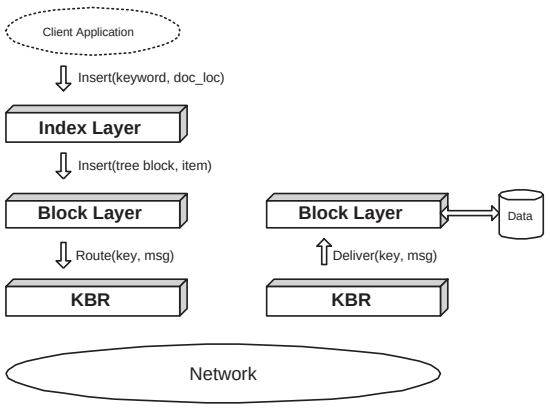

Fig. 1. The system is built with a base DHT overlay network for managing hosts membership in a scalable way. Each host of the system contains three components: a key-based routing (KBR) layer, the tree management (block) layer and the client index interface.

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

(a) Index Layer Interface

- BLOCK-ITEM-INSERT (blk_key,doc_loc)
- BLOCK-ITEM-REMOVE (blk_key,doc_loc)
- BLOCK-GET (blk_key): $blk\_contents$

(b) Tree Management Layer Interface

Fig. 2. Operation interface available at the Index and Tree Management layers.

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 [12].

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 document locations 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.

B. DHT Interface

The DEB Tree implementation uses a custom DHT interface that substitutes the typical GET/PUT interface with just a single *route* call. The operations offered by the DEB-tree implementation, 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 [13].

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, $\mathrm{ROUTE}(key, message)$, which is the case for all DHT implementations [14]. We assume the DHT layer will create a perfect balancing between DHT keys and peers through the use of additional balancing techniques [10], [11].

C. DEB Tree Algorithm

We will now describe the DEcentralized Balanced tree implementation [15]. This tree algorithm was based on the B^+-tree design and shares the high-availability requirements present on B-link trees [16]. However, unlike the B-link tree algorithm which was designed for a cluster based architecture with global system view and centralized environment, this 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.

Our algorithm enables the use of small sized blocks both for small or large sets. Small sets can be stored inside a single block whilst large sets will be distributed over a block tree, starting at the root block and appending a new block level as new information is inserted onto the set. DHT blocks contain either index occurrences (document locations) or references to other blocks. For every index keyword there exists always a root block, empty by default. Each block grows in size as occurrences are inserted. When a block reaches a reference limit size it creates new blocks and divide it's contents with the new blocks. Depending on the block size the DHT will possibly have to store more than one block per host if the number of blocks exceeds the number of hosts. We assume that the storage capacity of any host is sufficiently large to hold all the blocks that are assigned to it. Again, this is proportional to the indexing load injected in the system by these same hosts.

Blocks are distributed over the DHT through an hash value of the block key. To improve this initial load balancing strategy, dynamic techniques can be applied to the DHT in order to maintain the (block) keys evenly balanced. We expect that by using small sized blocks, load balancing will improve considerably.

IV. 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.

A. 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.

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. Removing an index occurrence works in the same way as for the insertion case.

B. 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 re-ordering.

The early-pruning heuristic was inspired by the adaptive set intersection algorithm suggested by Li et al. [17] 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

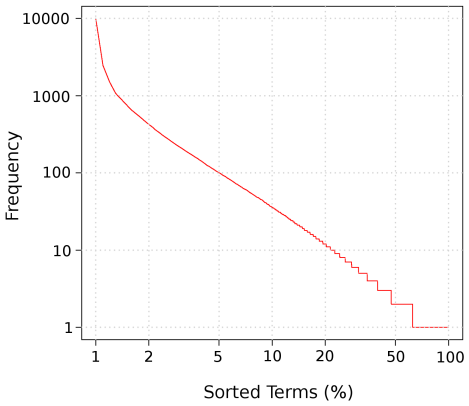

Fig. 3. Frequency distribution of sorted keywords (terms) in the textual collection.

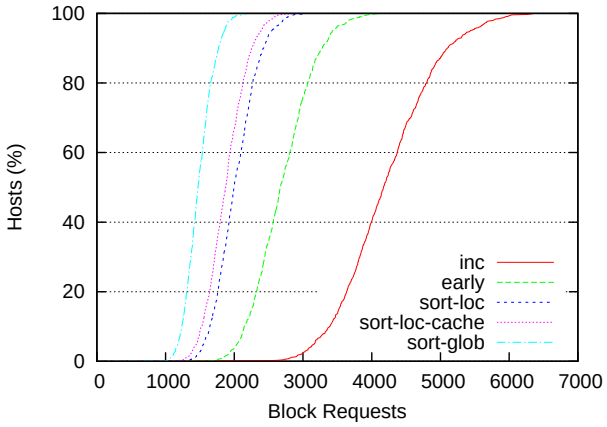

Fig. 4. A cumulative distribution function (CDF) of the number of block requests (messages received) on hosts for the different searching methods with a block size of 32 items.

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.

C. Internal Block Caching

Tree based structures require all operations to access the tree's root block. This access pattern creates a network hot spot on the host storing it. We overcome this limitation by caching the root and higher level blocks at clients. By using a B^+–tree algorithm, all client data is stored exclusively at leaf blocks. Since clients target only leaf blocks, internal blocks are used only for leaf block location within the tree. Caching internal blocks will not have any side effect to the success of client index operations as they operate exclusively over leafs. It will however decrease the number of block accesses a client has to make in order to locate the target leaf, and relieve higher level block contention, namely the root block when cached.

The larger a tree is, the less probability higher level blocks have of being modified and therefore becoming outdated on caches. On the other hand, the bigger a tree is, the higher probability one of it's top level blocks will have of being accessed and the higher chance a cached block will be used. For the trees that fit a single (root) block, internal block caching is not useful and clients will access the block directly.

V. Evaluation

We performed a simple evaluation of query methods using a custom made discrete event simulator implementing the DEB-tree algorithm over 1000 virtual hosts. The simulation ran queries over a collection of 10.000 documents with an average size of 3KB. The Figure 3 shows the distribution of keyword occurrences in the textual collection, depicting a Zipf distribution.

The Figure 4 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. A vertical line would represent a perfect uniform distribution.

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 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. 4 (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 [9].

VI. Conclusion

DHT systems provide scalable distributed data store solutions with efficient object location and key balancing among hosts. However, 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 inverted indexes.

In this article we bring attention to the issue of balancing storage and network resources among hosts and present a system that adapts classical and proven techniques, Balanced Trees, to a demanding distributed setting. The solution is capable of storing an inverted index and maintaining load balance between all hosts in the system.

We evaluated our algorithm on a concurrent simulated environment with a textual reverse index, a highly skewed dataset. The results show that query optimizations are capable of balancing network load distribution on hosts, improving the overall scalability.

REFERENCES

[1] A. Moffat, W. Webber, J. Zobel, and R. Baeza-Yates, "A pipelined architecture for distributed text query evaluation," *Information Retrieval*, vol. 10, no. 3, pp. 205–231, 2007.

[2] 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.

[3] 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*, 2001, pp. 149–160.

[4] 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 Conf. on Distributed Systems Platforms*, Germany, 2001, pp. 329–350. [Online]. Available: citeseer.nj.nec.com/article/rowstron01pastry.html

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content addressable network," in *Proceedings of the ACM SIGCOMM'01 Conference*, 2001, pp. 161–172. [Online]. Available: citeseer.nj.nec.com/ratnasamy01scalable.html

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

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

[8] 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.

[9] 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)*, March 2004, pp. 99–112.

[10] 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, USA, February 2003.

[11] D. Karger and M. Ruhl, "Simple efficient load balancing algorithms for peer-to-peer systems," in *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS'04)*, Berkeley, CA, USA, February 2004.

[12] 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.

[13] 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*, 2005, pp. 97–108.

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

[15] N. Lopes and C. Baquero, "Using distributed balanced trees over dhts for building large-scale indexes," University of Minho, Tech. Rep., October 2006.

[16] T. Johnson and P. Krishna, "Lazy updates for distributed search structures," in *Proceedings of the 1993 ACM SIGMOD Intl. Conf. on Management of data (SIGMOD '93)*. New York, NY, USA: ACM, 1993, pp. 337–346.

[17] 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.