Using Distributed Balanced Trees Over DHTs for Building Large-scale Indexes

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

DI/CCTC, Universidade do Minho, Portugal
October, 2006

Abstract

DHT systems are structured overlay networks capable of using P2P resources as a scalable storage platform for very large data 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 composable 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 [16, 14, 12] are scalable towards the total number of hosts, requiring only logarithmic storage space and network messages for locating any key. An hash function is used to uniformly distribute keys to hosts so that key load is balanced. However, even with a perfect key to host distribution, hot spots created by data or query asymmetries may occur. When a single key is accessed very often, a network bottleneck appears on the host storing that key. This situation known as "query flash crowd" can be minimized with caching schemes [15]. On the other hand, storage hot spots occur when very large objects of skewed size are stored on individual DHT keys. 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 cannot be eliminated by caching, here, 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 [8, 11] are also unable to eliminate the storage hot spots because the storage unbalance is due to a single key containing a very large object.

Several applications have been using DHTs as a scalable platform for storing application data [4, 18, 2, 17]. However, this application data is sometimes highly skewed, generating large data objects and inducing the aforementioned problems in the DHT storage. These large objects are either created from single indivisible objects or from grouping many smaller individual items. Single large objects are supported on file-system based DHT systems [4] by splitting the large file into multiple smaller pieces, storing each one as a DHT object. Modifications on the large file are done by creating DHT objects with the new data and appending these DHT keys to the history log of the file.

There are however other kind of applications where a very large object is made of multiple smaller independent items that must be treated individually. Consider, for example, a set structure which must not contain repeated items. This is the case when storing a textual inverted index directly over the DHT, where text keywords (DHT keys) are mapped to document reference *sets* (DHT objects) [10, 13, 18]. Because the distribution on the document reference set size typically follows a power-law distribution, some keywords can have a very large document reference set [21]. This large set, made of several document references, maps into a very large size DHT object which will cause a storage hot spot on the host storing the object. This problem is not unique to textual inverted indexes. In fact, other index data sets may also suffer from unbalanced load distribution, creating skewed size objects [2].

We propose a solution for load balancing DHTs when storing (decomposable) objects of non-uniform size. We developed a new DEcentralized Balanced tree (DEB) tree algorithm capable of converting a very large object into multiple bounded size pieces suited for being stored and searched as objects over DHTs. We used the DEB algorithm to build a textual inverted index that runs over any DHT implementation that provides a Key Based Routing (KBR) interface [5]. Our results show a more uniform distribution of storage and network resources over hosts. Although we only applied this algorithm for building a textual index, the algorithm offers the same functionality found on balanced trees including range-queries, making it also capable of building any generic (non-textual) index.

Our paper is organized as follows: Section 2 shows an overview of the system, Section 3 describes the DEB tree algorithm and Section 4 the textual index system. Section 5 shows our evaluation results, and Section 6 presents related work. We conclude in Section 7.

2 System Overview

We present a P2P indexing system as an application of our DEB tree algorithm. Although we use DEB trees for indexing text data, the algorithm offers a generic interface and could be used for any indexing data set.

- INSERT ($keyword$, $doc\_location$): $status$
- SEARCH ($keyword_{list}$): $doc\_location_{set}$

Figure 1: The global index interface for user applications.

In order to access the global index, a host must first join the system. After joining the system, the host contributes with storage and network bandwidth to the system while at the same time accessing the global index functionality. By using a DHT based structured overlay network, we leverage scalable routing and dynamic membership management.

2.1 Index Model and User Interface

A textual inverted index stores relations between text words (the vocabulary) and sets of document locations (the occurrences) [1] 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 that globally identifies a document by the host where it is stored and a local identification number. Other location schemes could be used, like an URL link or the document content hash value, provided the location of the document can be determined from the hash value [10, 17].

The index is made accessible to system peers through the interface on Figure 1. User applications contact the local index library through this interface either requesting the addition of a new document into the index or requesting for documents that contain a specific keyword (or keywords). The index insert operation adds a new relation between a keyword and a document. Clients are required to call this operation for each association they intend to insert on the global

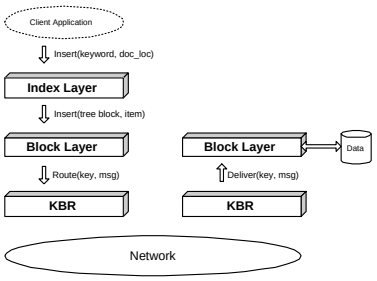

Figure 2: The system is built with a base DHT overlay network for managing hosts membership in a scalable way. Each host of the system contains three components: a key-based routing (KBR) layer, the block storage module and the client index interface.

index. The index search operation retrieves the list of documents associated with a keyword. We only considered the *and* Boolean Query operation for multiple keyword queries, although the remaining boolean operators could also be implemented [1]. The *and* operator applies the set intersection to all the keywords occurrence sets, returning the list of documents that contain all keywords.

2.2 Layered Architecture

The system architecture, depicted in Figure 2, is composed by the index layer that presents the index interface to client applications, the tree management layer that implements our distributed algorithm and the routing layer 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.

We store the index data, the occurrence sets, as DEB tree instances where each tree contains the document locations for a single index keyword. All DEB trees are independent from each other and stored side by side on the system. Since a tree is composed by many pieces, named tree blocks, distributed across several hosts, a single index operation may require the access to several tree blocks, and therefore, several different hosts. The calling host follows an iterative model, calling tree operations on the necessary

- BLOCK-INSERT (blk_key, $item$): $status$
- BLOCK-GET (blk_key): $block$

Figure 3: The tree layer interface for index operations.

tree blocks until the index operation is finally completed.

The tree layer implements the decentralized tree operations and is responsible for storing the block data on hosts. The tree layer interface to the index layer and for intra-block communication is depicted in Figure 3. All tree operations are block oriented, i.e., they require a block to work on and are executed locally on the host storing the block. This design differs from the two cycle invocation pattern, *get*-execute-*put*, found in other layered systems where clients fetch the entire block data, perform some data modification at the client and then write back the new data on the storing host [2, 20]. Although the *get*-execute-*put* design is compatible with the standard put and get DHT interface, it uses one more DHT access to complete the tree operation. This extra DHT operation not only increases latency, for an operation that could be otherwise executed on the server side, but also increases data conflicts when two clients access and modify the same block concurrently.

Our design avoids the concurrency issue by serializing operations on the server side (the host responsible for storing the block) while using only a single block access to execute the tree operation. Furthermore, executing block operations over a single operation request reduces latency by half to the client host. This design is not compatible with the standard DHT interface available on every DHT implementations, like the OpenDHT system for example. Instead, the DEB tree algorithm requires a key based routing interface that is present (either explicitly or implicitly) on all DHTs. If this routing interface is not available, the algorithm could also be adapted to run under the *get*-execute-*put* model with minor changes.

The Key Based Routing (KBR) layer main purpose is to route messages efficiently between hosts given a key. It's interface is defined by the single function:

ROUTE($key, message$), that given a key will deliver the $message$ to the host responsible for key [5]. Each tree block has it's own unique identifier, that can be used as the key parameter for this function. This function is therefore used by the index layer operations and by block layer management operations to send a message to a specific tree block, given it's key.

2.3 System Availability

Our system inherits the fault-tolerance offered by typical DHT systems. Churning influences negatively DHT routing and data availability. We assume the system is stable by filtering unstable hosts and creating a stable low churn environment. Unstable hosts may still access the index, by proxying their requests through the index layer of overlay peers. Clients can re-insert index relations to rebuild index data in case all replicas of a tree block (DHT object) fail. Tree internal dependencies can also be rebuilt from re-announces, which will be described on the following section. In case a block message sent from an index layer client is lost, messages are simply retransmitted after a time-out period.

3 Decentralized Balanced Trees

We will now describe our DEB tree implementation. This tree algorithm was based on the B^+-tree design [3] and shares the high-availability requirements present on B-link trees [7]. 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. Moreover, by choosing a block oriented algorithm, we could easily adapt it to a scalable storage DHT system which provides a key-based access to data while having a bounded logarithmic block access cost on the number of stored items.

3.1 Tree Structure

The tree structure, just like in the B^+-tree design, is composed by a root block and child blocks. Each block can have child blocks associated to it, which is called an internal block, or not, which is then called a leaf block. 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 that has themselves as a child block. To improve availability, each block also stores the key to the next sibling block, following the B-link design.

The size of 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 [3]. For internal blocks the degree influences the number of child block keys it contains. For leaf blocks it influences the number of data items the block stores.

In addition to the previous fields, each block contains the minimum and maximum limits, representing the interval of data the block is responsible for. The root block has an infinite interval, covering all data. Child blocks have a non overlapping subinterval such that the sum of all block intervals (defined by it's limits) from the same tree level equals the infinite interval. Internal blocks also store the limits of each child block such that the sum of all children block limits should be equal to the parent block limits.

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. Since DHT hosts cannot rely on any centralized entity, a new block identification must be generated from an already valid block key autonomously and be itself also globally unique. An additional restriction is that

since valid block keys are already in use, generating a new key from a previous valid key must not alter the previous key.

The block key must be unique within the tree to which belongs but also distinguishable from other trees. We achieve the uniqueness outside the tree by importing the index keyword as part of the key. The index keyword uniquely identifies the tree a block belongs to. Within the tree itself, each block is identified by the level of the tree and the minimum limit field. Since there cannot be two blocks with overlapping intervals, the minimum limit operates as the distinct value for blocks at the same level. The final structure for the identifier is the following tuple: $\langle keyword, level, minlimit \rangle$. The tree level is 0 by definition for the tree's root block. Leaf blocks always have the level 1. The remaining levels are incremented from the leaf level until the top. In order to keep track of the tree's overall level, the root block uses an additional integer field containing the current tree level.

Lets present a numerical example showing a new identifier generation from the following identifier $\langle abc, 3, 2 \rangle$, a block of the "abc" keyword at level 3 with the limits (2,8). The new identifier will be given a minimum limit value of 6, and so it's value would be $\langle abc, 3, 6 \rangle$. This example used numbers as limits but in our case, the limit values are of the same type as the data items stored on the tree, which would be document locations.

3.3 Index Client Requests

The operations requested by the client index layer on tree blocks are item insertions (or removals) and fetching data. These operations use the single block operation request pattern described previously to reduce the caller's latency. By placing the operation execution on the host storing the block, the operation must finish within the call context. In order to maintain high availability on data blocks these operations must never block or stop responding to clients, even due to background maintenance operations.

The insert item operation, described in Figure 4 receives a data item from the caller and the block into which the item should be stored. If the block is a leaf and the item is within the block's limits then the operation is successful. If the block is internal, a reply forwarding the caller to a child block key is sent back, provided there is a child block whose limits contain the item. The same is valid for the next sibling block. If the item is outside the block's limits and the block doesn't know of any other block capable of handling this item then an error message is sent back. The remove item operation is similar to the insert item except for the removal of the item and for launching the merge operation if the block size gets below the threshold value.

```
proc block-insert (blk-key, item):
block ← fetch block with key: blk-key from local storage
if item ∈ block.limits:
  if block is leaf:
    add item to block.data
    if size(block.data) >= block.tfactor*2:
      launch split(block)
    return ⟨ack⟩
  else:
    blk-child-key ← fetch child block key whose limits
      contain item
    ret message ⟨forward, blk-child-key⟩
else:
  ret ⟨error, item not within limits⟩
```

Figure 4: Pseudo-code for the block layer insert item function.

The fetch operation is used to retrieve the block's items to the caller. It's implementation is too simple and we'll skip it's presentation. It consists in fetching the block data from the local storage and return the block items (for leaf block) or the block child keys and limits (if internal) back to the caller.

3.4 Block Management Operations

Block management operations are used within the tree layer to create, split or merge blocks. These operations are called from inside a block or between two blocks. Again, the KBR layer is used to locate the target block and deliver the request message to it. These operations must run on the background causing minimal interference with front side operations,

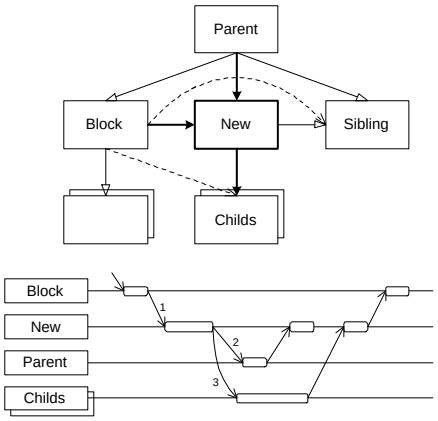

Figure 5: The splitting of a block involves creating a new sibling block, transferring half the contents to it and updating block references on the proximity blocks. Dashed arrows represent old references removed while highlighted arrows represent new references that are created during the operation.

the operations requested by the index layer.

When a block reaches it's maximum capacity, $2t-1$ items, the block starts a split operation that will divide it's items between itself and the new block. This is how the B^+-tree algorithm maintains all blocks with the same approximate storage load. The reverse would happen when the block reaches $t-1$ items, it's minimum capacity, where it would merge with another block and be removed; we did not address here the merging case as we don't expect trees to switch between a very large number of stored items into only a few items often. Although the B^+-tree algorithm establishes hard limits on the block size due to the secondary storage physical block size limit, our implementation does not impose strict limits and in practice tolerates transient limit overflows as storage is not strict on hosts. The block size value is local to each host, as the decision to split starts from the overloaded host. Each host could use a different block size according to the available storage and this value could even be dynamic during time. There is however a dependency between storage capacity and bandwidth use, as more data implies more network load.

The split algorithm considers two different cases: sibling split (that occurs on non-root blocks) and root block splitting. Non-root blocks split by creating a new sibling block and transferring half the contents into it. If the block is a leaf, only data items, i.e. document locations, will be transferred; on non-leaf blocks data includes child block references and their set ranges. Figure 5 shows a non-root block splitting where a block starts the operation by creating a new block id locally and sending a create block message request to the host responsible for the new key (message 1 in Figure). The host receiving the message will create a new block using the key and data provided by the splitting block. Although the block does not exist on the host, the host can establish that he is effectively responsible for such key using the KBR layer and create an empty block data before processing the message. It is the new block that will send a message to it's parent requesting to be inserted as a new child block (message 2 in Figure) and in case of being also an intern block, sending a message to each of it's children identifying itself as their new parent (message 3 in Figure).

After receiving confirmation messages from the parent and child blocks involved, the new block replies with an acknowledgment message to the splitting block and starts accepting client requests normally. Upon receiving the acknowledge message, the splitting block resumes normal operation.

While waiting for the new block to confirm the operation, the splitting block continues to process index operations in order to be responsive to clients. However, it will not start any concurrent split.

The root block splitting creates a new tree level, increasing the tree's height by one. Since the root does not have a parent block, the split operation is simpler and for space saving we omit the details.

3.5 Data Repairing

To tolerate host failures we need to replicate block data over a group of hosts. DHTs devise a simple form of data replication. Our application offers the same data consistency present on other DHT-based systems that depend directly on the DHT correctness properties. Because we assumed that our system is

mostly stable, we expect data consistency to be high.

In the event of block data loss due to the failure of all available replicas, we are able to partially reconstruct the block data. If the lost block is a leaf, data can be recovered by making clients refresh their data on the index. This refreshing is capable of reconstructing all the leaf blocks on the index. Internal blocks can be rebuild by making their children send a periodic announcement to the parent. By adding an extra field on each block, the maximum limit of the parent block, and including the parent's block key and the extra parent's maximum limit, each child can simply restore the structure properties of a parent block. As all children send their announcements, the parent's internal data, the child keys and their limits are fully restored. Each internal block is then partially replicated on it's children replicas.

However this recovery only restores a single parent level block, it cannot restore parents recursively unless all children are available. Another approach to be used in conjunction with the previous one is to reconstruct the tree from the root block. Assuming that all internal blocks of a tree were lost, including the root block, the tree could be reconstructed again by having leaf blocks or lower level blocks reinsert their references recursively on the root block and descend the tree until finding their parent block. The level field on the block keys could be used by the root block to determine the original tree size and create internal blocks at appropriate levels.

4 Textual Indexing System

We will now present our textual inverted index system. 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 on the tree layer to accomplish the initial index operation. Clients use an iterative model for accessing blocks, requesting a block level operation (listed in Figure 3) to be executed at some specific block and waiting for the reply before issuing another block request. Client block requests are atomic, in the sense that they depend only on the target block's state to produce an answer. This design increases block availability to clients because requests can be serviced as soon as they arrive to the target block's host. Clients have the responsibility to finish the index operation, i.e., calling all necessary blocks to conclude it.

```
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
```

Figure 6: Simple pseudo-code for the index insertion procedure.

4.1 Document Insertion

For inserting a document into the index system, peer clients use the INSERT (*keyword*,*doc_location*) function, which adds a document location to a keyword occurrence set. The client must call the INSERT function for every $\langle keyword, document \rangle$ 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 trees with more levels, the client must first find the correct leaf block, as items are only stored at the tree bottom level, by making a vertical traversal starting at the tree root block and following child block references. The operation terminates after locating the correct leaf block and receiving the acknowledgment of the insertion. Figure 6 shows a simple pseudo-code for the insertion operation on the client index layer. The block side code either replies with an *ack* message indicating the operation succeeded or with a *forward* message pointing the client to a child block (see Figure 4). When receiving a forward message the client reissues the insert request for the new block reference

recursively until receiving the final acknowledgment. The removal of a document location from a group of keyword occurrences is made just like for the insertion case, except that instead of adding the item, the item is removed from the block's data.

4.2 Multiple Keyword Search

Queries on index systems follow a multiple keyword distribution, using the *and* Boolean Query operator for returning the set of document locations that are common to all the query keywords Basically the operation applies the set intersection to all the keyword occurrence sets in order to obtain the final result set. To perform the set 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. The major disadvantage of this simple technique is requiring the client to retrieve the complete sets before applying the intersection operator. Fetching a complete large occurrence set uses network bandwidth to retrieve data that may not be necessary to effectively answer the query.

In order to prevent the full retrieval of large sets, we implemented an incremental approach based on a recursive breadth-first traversal of trees. Our incremental retrieval enables the use of two heuristics named early-pruning and term sorting to reduce the overall number of block access necessary to answer the query. Our objective in using an improved incremental approach instead of a simpler sequential approach is to reduce the network load imposed on the system while obtaining the same (correct) final result.

We implemented the incremental retrieval in the following way. Data items (in this case document locations) are stored on blocks according to the block's limits, that is, each block is responsible for a piece of the data domain. All blocks share the data limit concept, either internal or leaf. If the block is a leaf, it will store data items contained inside it's limits. If the block is internal it will contain references to child blocks whose data is also contained inside the block's limits also. The algorithm starts by gathering the root block keys of all keywords and place them inside a tuple containing the complete data domain interval $]-\infty, \infty[$. Note that by definition, the root block's limits are equal to this domain interval. Then the algorithm starts iterating over the block list and fetching blocks using the BLOCK-GET operation one at a time. If a leaf block is found, the algorithm stores the intersection of the block's data items with the already fetched items, initially empty. If the block found is internal, the algorithm creates new sub-intervals from the current interval in use (initially the complete data domain), one for each child's limit interval. These new intervals keep the blocks to visit, leafs and item information already gathered from previous accesses, as long as they are contained inside the new sub-interval. The algorithm repeats this procedure recursively, accessing blocks and creating more sub-intervals until no more blocks are left to visit and all leafs were accessed. The final result set is the union of each sub-interval items.

Li et al. [9] proposed the use of an adaptive set intersection algorithm to reduce the amount of network communication used to calculated the intersection of sets. Inspired by the adaptive set intersection proposal and the fact that the intersection for an empty set with any other set will always be empty, we devised an heuristic called early-pruning to reduce the number of block accesses when using the incremental retrieval. The heuristic basically stops further retrieval of blocks for a specific sub-interval if at least one leaf block was accessed and the temporary set is already empty. This heuristic is effective in selecting the branches of large trees to visit according to items already found on smaller trees while pruning the remaining branches.

Another interesting technique used by database engines to reduce the number of data access when determining the intersection of sets is term re-ordering. Term re-ordering consists in selecting the order by which sets are evaluated (or accessed), starting from the smaller sets and then proceeding into the larger ones. We implemented the term sorting heuristic by changing the order in which root blocks were accessed initially on the incremental algorithm. Term size was determined by the number of levels each keyword tree had, since each host does not have a global view of the index. This knowledge was gathered from previous accesses to keyword trees and used locally at the

index layer.

4.3 Internal Block Caching

All the previous operations require at least one access to the root block of trees. The index insert operation targets the leaf block responsible for the data item to be inserted and query operations make full breadth-first traversal of trees, both starting from the tree root block. This pattern creates a network bottleneck on hosts storing root blocks and higher level tree blocks. We address such limitation by caching internal top level blocks at clients. Caching internal blocks allows the client to determine the target leaf block locally without having to traverse vertically the tree and therefore reducing the network load placed on top level blocks.

As internal blocks are used exclusively for locating relevant leaf blocks, which is where index operations are effectively executed, caching does not interfere with the outcome of normal index operations besides reducing the number of accesses made by index clients. Furthermore, the larger a tree is, the less probability higher level blocks have of being modified and of becoming outdated on caches. We do not cache leaf blocks, as they contain the real tree data and are subject to modifications by index insertions.

5 System Evaluation

We will now evaluate the DEB tree algorithm implementation on a textual document collection. The algorithm offers two basic operations: document reference insertion and keyword search. The evaluation focus on the scalability of the solution when compared to the equivalent linear DHT mapping for 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. Our simulator used the co-routine python extension to simulate concurrency without any threads library. The communication between hosts and message routing, the Network and KBR layers, were simulated. Although our experiments ran over this network simulated environment, the algorithm implementation could be placed on top of a real DHT system for deployment. We opted for the simulation model to test the algorithm under a controlled environment with a larger number of hosts.

A small collection of text news documents was used as the data set for the simulation. The collection is made of about 10000 documents with 3kb size each on average.

5.2 Index Insertion

The simulation of the insertion procedure consisted in 1000 hosts inserting references for all the documents concurrently on the index. Each host was given 10 unique documents to insert. The format used to represent the document reference on the index was the object $\langle host\_address, docId_{local} \rangle$, containing the unique identification of the host where the document is stored and a document identification.

The primary purpose of DEB trees is to balance the storage load across hosts. We evaluated the algorithm performance by changing 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. This very large 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. The number of blocks inside each share is dependent on the hash function, but assumed to be evenly distributed. Figure 7 shows the cumulative distribution function for the storage load. We represent storage load as the number of stored items inside blocks. 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

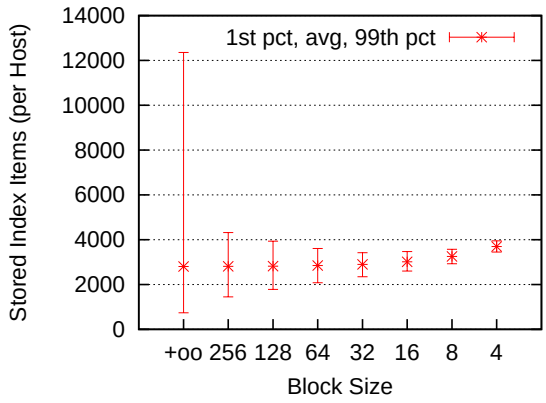

Figure 7: The storage load distribution (the 1st percentil, average and 99th percentil of the number of stored items) on hosts for different block sizes.

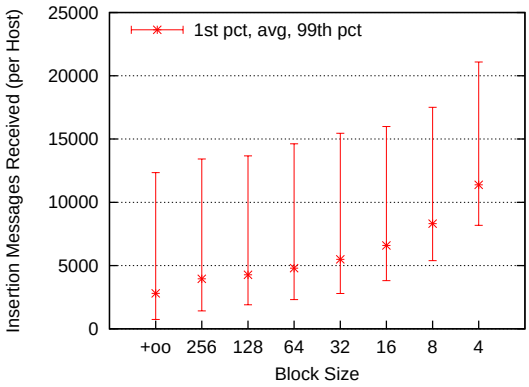

Figure 8: The network load distribution (the 1st percentile, average and 99th percentile on the number of messages received) for different block sizes when inserting 10k documents into the index *without cache*.

sizes tend to over-split trees, creating excessive internal blocks and increasing the load on the system. This is the case of the tree with block size 4. The best balanced and smallest average load tree has block size 32.

The Figure 8 shows the number of insert messages received by hosts for different block sizes. The direct mapping case (block size equals $+\infty$) shows a high variation between the minimum and maximum loaded hosts. This is due to large objects that receive much more messages than smaller ones and overload some hosts, while hosts that group smaller blocks receive far less messages. As the block size starts to decrease, the variation continues to be present, although at a higher load average. This happens because smaller size blocks split more often and increase the tree height. As the DEB tree requires a vertical traversal of the tree for every item insertion, more blocks have to be accessed to complete the operation, one block per tree level. The Figure 9 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 simulation with the very large block size, containing only single root block trees, is identical to the pre-

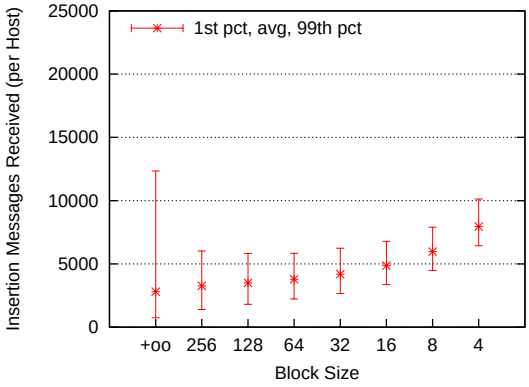

Figure 9: The network load distribution (the 1st percentile, average and 99th percentile on the number of messages received) for different block sizes when inserting 10k documents into the index *with cache*.

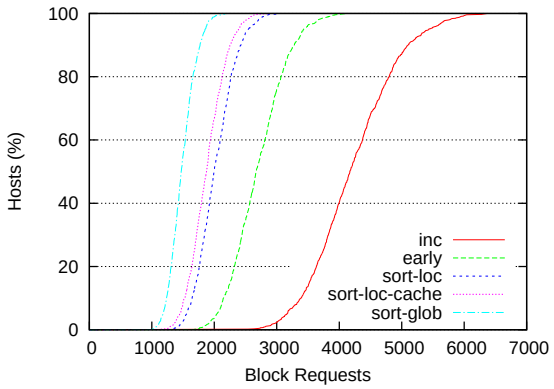

Figure 10: A cumulative distribution function (CDF) of the number of block requests (messages received) on hosts for the different searching methods with a block size of 32 items.

vious 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.

5.3 Index Searching

The index searching procedure starts from a fully loaded index on the system. A single host process a list of multiword queries issuing getBlock operations on several tree blocks and gathering the results. We generated 20000 queries with a multiple keyword distribution from the original text collection set. Each individual keyword follows the same frequency distribution found on the document collection. We also shuffled the query keyword frequency so that popular keywords on documents would not match popular keywords on queries. The shuffling prevents the algorithm from benefiting with the same distribution on both inserts and queries. Search performance was measured as the load distribution imposed on hosts. We counted the load as the number of index items replied to the caller.

We will first show the impact of our query optimizations on the system bandwidth. Figure 10 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. Then, 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 but before retrieving all tree blocks if possible. The term reordering method was originally developed for global knowledge, so we also simulated a variation (label `sort-glob`) that supplied the client with the system 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.

Finally we look at the impact of caching for the query methods on Figure 10. We implemented the reference cache procedure over the local term reordering heuristic (label `sort-loc-cache`). When comparing it to the same query method without cache (label `sort-loc`), one observes that although cache reduced the overall load, it was only marginally and it had not removed the highest load some hosts received. This performance can be explained by noticing that this cache was operating only on internal blocks, having no effect on leaf accesses. As queries follow a skewed distribution, it can happen that frequent accesses to (the same) popular keywords (for queries) require the retrieval of leaf blocks so often that create the previous host load asymmetry. Since this cache method does not cache leafs, 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 solution for this query "flash crowd" was already addressed in [15] and could also be implemented on the index client.

Figures 11 and 12 shows the distribution of the network load on hosts as function of the block size.

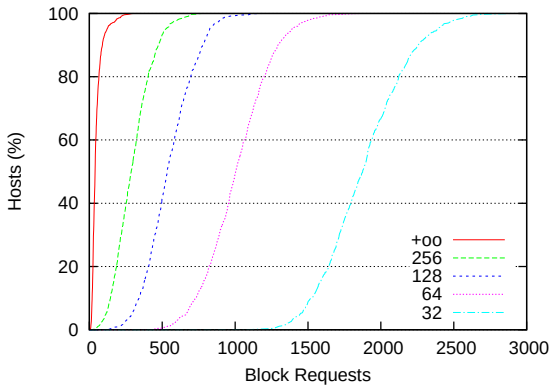

Figure 11: A cumulative distribution function (CDF) on the number of block requests (messages received) on hosts for the different block sizes using the local term re-ordering search method with cache.

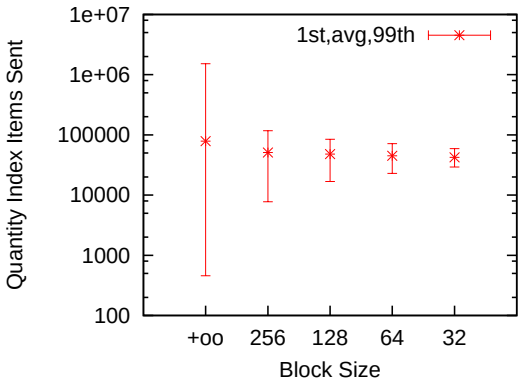

Figure 12: The network load distribution (the 1st percentile, average and 99th percentile on the quantity of index items replied to the client) on hosts. The YY axis is in logarithmic scale.

We simulated the execution of the query data set described before using the incremental query method with early-pruning, local term re-ordering and reference cache for different block sizes and the direct DHT mapping case ($+\infty$). The Figure 11 shows the CDF on the number of block request messages received on hosts while the Figure 12 shows the 1st and 99th percentiles and average on the quantity of items replied to the client per host. The direct DHT mapping case shows a small number of requests received per host, however because all data is contained within a single block, the load difference between the 1st and 99th percentile varies within three orders of magnitude. As the block size decreases, the number of messages received at each host increases but the variation between the percentiles is diminishing, contributing to a more uniform load distribution on hosts. The number of messages received by hosts increase because blocks have smaller size, storing less data and causing tree splits, creating more blocks. Larger trees force the client to access more blocks to retrieve data. However, 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 transmitted back to the client). This figure also shows that our algorithm uses slightly less network bandwidth on average and with a better uniform distribution across hosts to execute 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 searching and tree based algorithms. Keyword searching systems use an index to store the relations between keywords and document references (also known as the occurrence sets). The index can be local to each host (partition-by-document) or shared among all hosts, each containing a part of the global index (partition-by-keyword) [9]. Our algorithm assumes a partition-by-keyword design and we will only compare it to other systems using the same design. Overcite [17], a P2P implementation of the Citeseer system with keyword searching, uses a partition-by-document design for their search functionality. The major difference with the two designs is that partition-by-document requires client searches to contact all peers while partition-by-keyword only requires contacting the host containing the keyword part of the index.

Previous work on textual inverted indexes over

DHTs did not handle the storage hot-spot problem we have identified [10, 6, 13]. They store the inverted index directly over the DHT, causing the storage unbalancing of some hosts that also results in network bottlenecks. Our algorithm creates an additional layer that automatically balances the storage load and consequentially the bandwidth used for creating the index over the DHT hosts. Tang and Dwarkadas [18] 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 items for the same index key through a fixed interval of DHT keys. This constant factor distribution does not take into account the final object size forcing clients to access all DHT keys inside the interval to manipulate data for the index keyword. Our algorithm adapts dynamically to the object size, ensuring a uniform storage distribution among hosts whether the object is small or very large.

We will now compare our work to other tree based P2P systems. Brushwood is a system that builds a distributed tree over a structured overlay to store data with locality properties [19]. Brushwood is bound to the Skip Graphs routing algorithm and just like the previous systems it does not handle the storage hot-spot problem, where a very large key would always be stored on a single host, overloading it. The main difference between Brushwood and our DEB tree implementation is that our algorithm is not bound to any particular DHT implementation, but instead relies on the basic Key Based Routing interface, which is present over all DHTs. There are other systems that just like Brushwood are also bound to a single DHT implementation, restricting the scalability of their solution to the underlying DHT implementation.

Chawathe et al. proposed the use of a Prefix Hash Tree (PHT) for building a layered index structure over a generic DHT for storing (x, y) coordinates of wireless access points [2]. Tree nodes are identified by a prefix which is taken from the data inserted into the tree. However, all the data assigned to the same item identifier (and consequently the same prefix) is stored under the same DHT object, leading to the storage hot spot issue. The main difference of the PHT to our algorithm is that our structure is not sensible to skewed data. The PHT, in order to adapt to data variations, places leaves at different tree levels, but even so it places items with the same prefix under the same DHT object. Our tree blocks adapt dynamically to the identifier distribution and always create a balanced tree, i.e, a tree where all leafs are at the same tree level and theoretically have a uniform storage size distribution. We also differ on the overall system design in which PHT uses a single system-wide tree to store the index while our system stores the index over many (smaller) trees.

Zheng et al. presented a Distributed Segment Tree (DST) algorithm designed to support range-queries over a generic DHT [20]. Range queries are queries that should retrieve all data items that fall within a specified identifier interval. Tree blocks are identified by static range limits over the identifier space. Static range limits are incapable of handling skewed data identifiers properly. For example, if many data items have the same identifier, or are contained inside a very small identifier space, they all will be assigned to the same tree node and consequently the same DHT object, creating a storage hot spot at this DHT object. The main difference of DST to our algorithm is that our node range is defined dynamically according to the data distribution and tend to create the best storage distribution, even with skewed data. Although we are not presenting our algorithm as a range-query structure, it shares with DST the basic structure to be able to execute them.

7 Conclusion

We have identified a problem over P2P indexing systems, the storage hot-spot, that while not being restrictive in itself (today's storage is not a limitation on most settings) creates additional network load that limits the current systems scalability. We implemented a Decentralized Balanced Tree algorithm capable of converting very large objects made of multiple items into bounded sized pieces suitable to be stored over any DHT system using a Key Based Routing interface. By designing an algorithm that relies on a commonly available DHT interface, we were not bound to any particular implementation and are able

to choose the most suitable DHT according to our requirements on memory/routing usage.

Our algorithm was based on Balanced trees which are used by database engines to index data. Again, the decision to base our algorithm on a well tested structure let us benefit from previous research on query optimization techniques and efficient generic index functionality (like range-queries) that were already available on the B-tree design and are also present on our algorithm.

We evaluated our algorithm on a concurrent simulated environment with a textual distributed index system to determine it's balancing properties on both storage and network resources for a highly skewed data set. 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 for 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 algorithm, with query optimization techniques, to the direct DHT case.

Acknowledgments

We wish to acknowledge the insightful comments and suggestions made by Sylvia Ratnasamy and Rodrigo Rodrigues under the 1st ACM Eurosys Authoring Workshop.

References

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

[2] 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.

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

[4] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating System Principles*, Alberta, Canada, October 2001.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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