DINX: A Decentralized Search Engine

Blaise Gassend, Thomer M. Gil, Bin Song
MIT Laboratory for Computer Science
{gassend,thomer,bins}@mit.edu

http://dinx.lcs.mit.edu

Abstract

We present DINX, a search engine for peer-to-peer storage systems. DINX lets users do keyword searches. Most peer-to-peer systems do not have an efficient decentralized search engine. DINX fills this void.

The main challenge is partitioning the index. Naively partitioning the entire index by document over different nodes is potentially inefficient because it may force clients to query all nodes. DINX also maintains a distributed index partitioned by document, but increases the likelihood of finding satisfying matches from a small subset of nodes by indexing a file on a number of nodes proportional to some measure of its popularity, e.g., download count. This replication policy is based on the assumption that only a small fraction of all files is very popular. DINX clients first send a query to a single randomly chosen DINX server, and then collect results from an increasing number of servers if the user is unsatisfied with the returned matches. We have implemented DINX to index Web pages on top of a simple peer-to-peer network.

1 Introduction

This paper presents DINX ("Distributed INdeXer"), a decentralized search engine for peer-to-peer storage systems. DINX lets users do keyword searches.

It seems hard to achieve both efficiency, searchability, and decentralization within a single peer-to-peer storage system. Napster [14] has an efficient, but centralized search engine. This makes it vulnerable to lawsuits and outsiders who wish to control content. Gnutella has a decentralized search engine that works by flooding queries, which makes it very inefficient. Recently developed peer-to-peer storage systems such as Freenet [4, 7] and CFS [5] have no search engine at all.

Naively partitioning the entire index by document over different nodes is inefficient because it often forces clients to query all nodes. DINX also maintains a distributed index partitioned by document, but improves this by using extra available disk space on nodes to index certain files more than others: a file is indexed by a number of nodes proportional to its popularity, e.g., download count or reference ("link") count. DINX search queries first consult the index on one randomly chosen node and then the indices on an increasing number of nodes if the user is unsatisfied with the returned matches. DINX' replication policy increases the likelihood that querying a small fraction of all nodes suffices to find satisfying matches.

We have implemented a simple version of DINX that indexes Web pages on a system with properties similar to Chord [3, 15].

We believe that systems such as Chord, Freenet, and Gnutella could benefit from using DINX.

The rest of this paper is organized as follows. Section 6 takes a look at related work, Section 2 explains the design of DINX, Section 3 looks at the details of a DINX implementation, Section 4 presents the results of measurements on a DINX prototype, Section 5 discusses future work, and Section 7 concludes this paper.

2 Design

DINX maintains a distributed index partitioned by document. Each file is indexed on a subset of all nodes, the size of which is proportional to the file's popularity. DINX adjusts the ratio between the size of this subset and the page's popularity to fully use available disk space.

There are two types of DINX nodes: *indexers* and *managers*. An indexer indexes files; a manager is responsible for managing index meta-data and for telling indexers to index certain files. One node can be both manager and indexer.

Each file has two scores associated with it: a *popularity score* and a *relevance score*. A popularity score is the measure for a file's popularity, e.g., number of downloads per day. The number of indexers that index a file is proportional to the file's popularity score. This increases the likelihood that querying a small number of indexers results in a satisfying answer. This replication scheme is based on the assumption that only a small fraction of all files is very popular.

The relevance score of a file is a measure for its applicability given a certain query; it determines the set (and the order) of matches returned to a querying client.

A file's type determines the scoring methods. Number of downloads per day may be good for scoring audio files. The number of links to a certain page is a good method to score Web pages—Google [1] demonstrates this. DINX allows different scoring methods for different file types.

2.1 Indexer

An indexer has two tasks: maintaining an inverted index (word to URL-list mapping) on disk and answering queries from clients. When queried by a client, an indexer computes a relevance score for all files in which all keywords from the query occur and returns this list sorted on a combination of relevance and popularity score, the *combined score*. Section 3.3 deals with this issue in more detail.

2.2 Manager

Each file has a single manager associated with it. The details of assigning a manager to a file depends on addressing in the underlying storage system. (An example is a hash function that maps a URL to an IP address.)

A manager is responsible for storing index meta-data, for computing a file's popularity score, for telling indexers to index a file, and for retracting index entries from indexers when a file is no longer available. The exact number and location of indexers that index a file depends on the file's popularity and the available disk space in the underlying storage system. Section 3.5 deals with this issue in more detail.

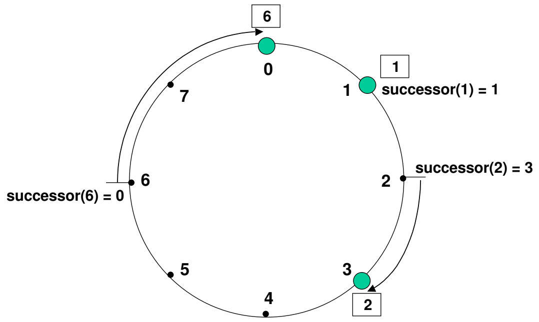

Figure 1: A Chord ring with three nodes 0, 1, and 3. The circles are nodes. The arrows illustrate Chord's successor function: a key (square) is mapped onto the first node that follows its hash on the Chord ring.

3 Implementation

In this section we describe the implementation of DINX which indexes Web pages on top of a simplified version of Chord [15]. This implementation does not include a mechanism that automatically adds new Web pages. However, we intend to build a crawler or implement some notification system into the storage system itself.

Unfortunately, we were not able to produce measurements to evaluate our system. This is due to the absence of a good testing environment and time constraints.

3.1 Chord

Chord consists of connected nodes that collectively implement a peer-to-peer system. Each node has a Chord-ID that uniquely identifies it. The Chord-ID space consists of 160-bit numbers laid out consecutively around an imaginary ring ("Chord ring"). Each participating node can be thought of as located on that Chord ring. A single node can have multiple Chord-IDs and, therefore, be located on different locations on the Chord ring. The *successor* of a certain Chord-ID k is the node with a Chord-ID that is equal to or follows k. This is based on consistent hashing [10]. Figure 1 illustrates this.

The *distance* between two nodes a and b is the number of nodes between a and b counted clock-wise along the Chord ring.

3.2 Overview

We implemented DINX on top of a simplified version of Chord that does not allow nodes to join or leave the ring. In addition, each DINX node has a list of the addresses of all other DINX nodes. In this implementation, every manager also runs an indexer.

DINX was mostly written in C++ using Berkeley DB and the `async` library that comes with SFS [11]. Inter-node communication is done through asynchronous RPC.

We have built a Web interface (available at `http://dinx.lcs.mit.edu`) for DINX. DNS could be configured to map requests to `dinx.lcs.mit.edu` to several different machines in round-robin fashion to avoid having a single point of failure. Through the Web interface users can query more nodes when results are unsatisfactory.

3.3 Indexer

An indexer manages several tables: `Word2ID` maps each keyword to a `wordID`, `InvertedIndex` maps each `wordID` to a set of `pageID`s, and `PageTable` maps a `pageID` to a page's URL, its title, its popularity score, its usefulness (explained in Section 3.5), and the file contents as a list of `wordID`s.

When queried, an indexer maps the keywords in the query to sets of `pageID`s (via `Word2ID` and `InvertedIndex`), computes the relevance and the combined score for each `pageID` in the intersection of these sets, and returns a list of {title, URL, combined score} tuples. An indexer computes the relevance score of a page using the number of occurrences of the search terms in the page and proximity of search terms to each other in the page. This could be augmented with techniques described in [1, 2, 13].

`InvertedIndex` is implemented using a set of files $(F_0, F_1, \ldots, F_k)$; file F_n contains records of size 2^{n+1} bytes that store lists of length l bytes, where $2^n < l \leq 2^{n+1}$. (In other words, lists of similar length are stored together in one file.) A separate file, memory-mapped as an array, maps `wordID`s to $\{i, r\}$ tuples, such that the r-th record in F_i contains the desired list of `pageID`s. This setup allows `InvertedIndex` to be updated incrementally: lists can grow within a file without having to move data that follows it. Modifications to files are first cached in memory and committed only when the cache is full or on explicit instruction. This is to minimize disk access.

3.4 Manager

DINX finds the manager of a page by applying the Chord successor function to the file's URL. When some outside mechanism, e.g., a crawler, notifies the manager of some new or changed page, the manager fetches that page and assigns it a popularity score. A page's popularity score is its *link count*, i.e., the number of links to that page from other pages. The manager then instructs a number of indexers to index the page. Section 3.5 explains this in more detail.

Concretely, a manager's database consists of 2 tables. For each managed page the `References` table stores the list of URL hashes of pages that link to that page. (Links to non-existent pages are stored in a separate table.) In addition, for each managed page the `FileData` stores the page's URL hash, the URL itself, a list of URL hashes of links in the page to other pages, the page's popularity score as reported most recently to an indexer, the time of that report, and the Chord distance to the furthest indexer that indexes the page. `References` is implemented similar to `InvertedIndex`.

The motivation to maintain table `References` and not simply the link count is the possibility to implement different mechanisms that maintain consistency in face of networks failures. To keep `References` up-to-date, managers inform each other of adds, deletes, and updates of pages. These mechanisms require l messages from manager to manager for an added or deleted page with l links. Managers notify indexers of a new popularity score when the popularity score has halved or doubled, or when 24 hours have passed since the last notification. This piggybacking of notifications reduces the amount of traffic between DINX nodes. In Section 5 we describe a technique that simplifies these mechanisms.

3.5 Replication

The number of indexers that index a page is proportional to its popularity score. We define the *usefulness* of indexing a certain page on a certain indexer to be $u = s/(d+1)$, where s is the page's popularity score and d is the Chord distance between the indexer and the page's manager. In other words, the usefulness

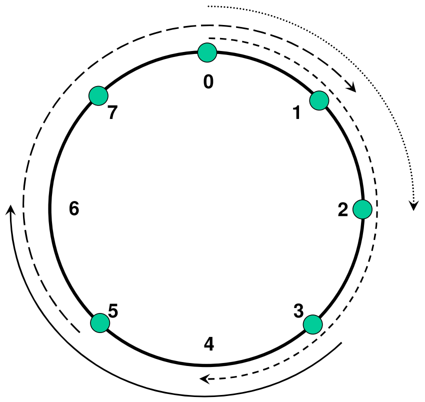

Figure 2: Replication in DINX. An arrow represents the segment of the Chord ring on which the page is indexed. The segment size is proportional to the page's popularity.

of indexing a page on an indexer is proportional to the page's popularity score, and inversely proportional to the distance from its manager.

A page's manager estimates how far along the Chord ring the page should be indexed. This estimate is based on the page's popularity score and the lowest u in the local database. (Chord's load-balancing property guarantees that this u is roughly uniform across the ring.) Consequently, pages with a higher popularity score cover larger segments of the Chord ring, which increases the likelihood of finding them when querying only a small subset of nodes. Figure 2 illustrates this.

When an indexer exceeds its disk capacity, it eliminates entries in the index for pages with lowest usefulness. If the last indexer that indexes a page does this, the page can no longer be found. However, the page's manager always retains a reference so as to reinstate the page if its popularity score increases. Because of its implementation, deleting an entry from `InvertedIndex` has the same cost as adding an entry.

3.6 Queries

A designated client, `dinxdo`, performs DINX queries on behalf of users. `Dinxdo` first passes all search terms to one randomly chosen indexer. A user who is not satisfied after inspecting the results instructs `dinxdo` to try harder. `Dinxdo` then queries twice as many nodes, chosen to be far apart from each other. Since pages are indexed on neighboring nodes, this mechanism ensures that `dinxdo` queries nodes with minimally intersecting content.

4 Measurements

Measurements presented in this section are rough performance estimates of the DINX implementation described in the previous section. Absence of a realistic testing environment and time constraints made it impossible to evaluate DINX in conditions that approach its target environment.

Rather than running hundreds or, preferably, thousands of machines, 4 machines ran one DINX manager/indexer combination ("server") each. (Some tests involved less servers.) Client programs that initiated work ran on separate machines to avoid competition over resources on the server machines. However, other users used server machines during measurements, which introduced noise in the results.

We indexed 15,000 pages from the MIT Web site [12]. Disk quota was set to 10 MB per DINX server. This limited disk space enabled us to observe behavior when servers run out of disk space and can no longer index all pages.

4.1 Database

Tests to measure indexing speed were done using a designated program that iterates through a list of URLs and, one by one, instructs 1 of the running servers (randomly chosen) to index that page. Doing multiple index requests in parallel may improve the presented numbers.

Currently, DINX is slow. The graph in Figures 3 shows the average time to index a single page in a 2-DINX server system given the number of pages already in the database. These measurements were done given a disk quota of 1 MB per machine. After having indexed roughly 400 pages, the database reaches its disk quota; inserts become more expensive as they are preceded by delete operations that free up entries of pages with lower usefulness. This extra overhead is clearly visible in the graph. After that, the cost per page drops again: as the lowest usefulness in the table increases, more and more insert requests are rejected.

We expect the cost to index large volumes of pages to decrease with more available servers since, by virtue

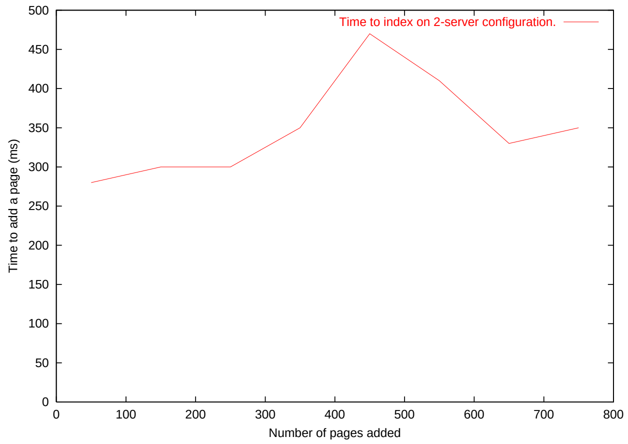

Figure 3: Time to add a page given the number of pages already in the database.

Servers	Add	Add (quota)
2	7:08	[[X]]
3	6:30	15:48
4	8:41	16:19

Figure 4: Times to index 1000 pages with and without imposed disk quotas

of DINX' replication mechanism, a decreasing number of servers will index a page as the system fills up.

Without replication, Web pages with an average size of 11 kB per page consume roughly 2 kB per page in the database. With replication, DINX naturally uses up its entire disk quota.

The low indexing speed that we measured is most likely the result of the overhead incurred by the indexer repeatedly forking a Perl script that fetches a Web pages and preprocesses it for indexing. We expect to make significant performance gains by implementing this functionality in the indexer itself; this eliminates an expensive fork per page. Without using the Perl script, indexers can build a 150 MB large reverse index for 90'000 prefetched and preprocessed Web pages at a rate of roughly 100 pages/second.

The table in Figure 4 shows measured times to index 1000 pages on a 4-DINX server system. Column "Add" gives the times to add pages with no imposed disk quota. Column "Add (quota)" gives the times to add pages to a database already using its entire disk quota. In the latter case, DINX has to delete entries from the database before adding new ones.

4.2 Queries

Because of the limited database size and the rather coarse scoring methods, DINX does not give outstanding results to queries. However, we observed some interesting behavior. For instance, the contact page for the Webmaster got high scores because it is referenced by almost all other pages. We expect that filling the database with more pages from many different Web sites will dampen out exceptions like these.

The choice of keywords strongly influences the response time for a query. Searches on more frequently occurring keywords take more time. Queries done through the Web interface at `http://dinx.lcs.mit.edu` have a latency of roughly 1 second for searches on rare keywords, and up to 3 seconds for searches on words such as "the" or "a". Bypassing the Web interface by using `dinxdo` from the command line greatly reduces latency: 30 ms for "daring", 1.2 s for "the".

5 Future work

We intend to make DINX more fault tolerant by dealing with joining and leaving servers.

We plan to extensively measure different parts of our implementation such as indexing speed and query speed. We intend to measure how many machines we would need to index a large number of pages and the resulting overhead. Based on the results of our measurements we will potentially redesign or tweak different parts of the system.

Replication currently depends on a page's link count. Keeping this number consistent introduces a fair amount of overhead. To simplify this, we intend to design a mechanism that replicates a file based on the number of actual file accesses. For example, DINX could automatically replicate files that a user accesses through the DINX search results page. This incorporates the actual user behavior into the replication mechanism.

We intend to experiment with multiple servers sharing a common database on a single machine to improve load balancing.

We plan to implement a mechanism that indexes highly popular files on a particular subset of nodes only.

We expect that this will greatly decrease overall traffic.

Finally, we will work to improve our ranking mechanisms.

6 Related work

The Inktomi (HotBot) [6, 9] partitions the index (by document) as disjoint sets over multiple machines. DINX, however, distributes the index as possibly overlapping sets since a popular page may be indexed by more than machine.

Gnutella [8] has a search engine that floods queries to other nodes. DINX, however, queries a single node and only queries more nodes if the user is not satisfied with the results.

The technique that DINX uses to compute the popularity of a file based on link counts is a simplified version of a technique used by Google [1].

7 Conclusion

We presented the design and a simplified implementation of DINX, a search engine for peer-to-peer storage systems that lets clients do keyword searches.

DINX increases the likelihood of finding satisfying matches from a small subset of nodes by indexing a file on a number of nodes proportionally to some measure of the file's popularity, e.g., download count. This increases the likelihood to find satisfying matches when querying only a small subset of all nodes.

Coarse measurements of a DINX prototype are promising, but indicate that there is room for improvement.

We believe that existing peer-to-peer storage systems would benefit from incorporating DINX.

8 Acknowledgements

We wish to thank Robert Morris, Frans Kaashoek, and all other members of PDOS for their input.

References

[1] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. Technical report, Computer Science Department, Stanford University, Stanford, CA, 1998.

[2] J. P. Callan, Z. Lu, and W. B. Croft. Searching Distributed Collections with Inference Networks. In E. A. Fox, P. Ingwersen, and R. Fidel, editors, *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 21–28, Seattle, Washington, 1995. ACM Press.

[3] Chord. The Chord Project, November 2001. Available at `http://www.pdos.lcs.mit.edu/chord/`.

[4] I. Clarke, O. Sandberg, T. W. Hong, and B. Wiley. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, California, July 2000. Available at `http://freenet.sourceforge.net/index.php?page=icsi-revised`.

[5] 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 Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[6] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-Based Scalable Network Services. In *Proceedings of SOSP '97*, St. Malo, France, October 1997.

[7] Freenet. The free network project, November 2001. Available at `http://freenet.sourceforge.net/`.

[8] Gnutella. Gnutella, November 2001. Available at `http://gnutella.wego.com/`.

[9] T. Grabs, K. Böhm, and H. Schek. A Parallel Document Engine Built on Top of a Cluster of Databases; Design, Implementation, and Experiences. Available at `http://citeseer.nj.nec.com/grabs00parallel.html`.

[10] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols

for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[11] D. Mazières. A toolkit for user-level file systems. In *Proceedings of the USENIX Annual Technical Conference*, Boston, Massachusetts, June 2001.

[12] MIT. Massachusetts Institute of Technology. Available at http://www.mit.edu.

[13] A. Moffat and J. Zobel. Information Retrieval Systems for Large Document Collections. In *Text REtrieval Conference*, 1994.

[14] Napster. Napster, November 2001. Available at http://www.napster.com/.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.