Distributing Google

Vijay Gopalakrishnan, Bobby Bhattacharjee, Peter Keleher
Department of Computer Science, University of Maryland
{*gvijay, bobby, keleher*}@*cs.umd.edu*

Abstract

We consider the problem of wide-area large-scale text search over a peer-to-peer infrastructure. A wide-area search infrastructure with billions of documents and millions of search terms presents unique challenges in terms of the amount of state that must be maintained and updated. Distributing such a system would require tens of thousands of hosts leading to the usual problems associated with node failures, churn and data migration. Localities inherent in query patterns will cause load imbalances and hot spots that can severely impair performance.

In this paper, we describe an architecture for constructing a scalable search infrastructure that is designed to cope with the challenges of scale described above. Our architecture consists of a data store layer which is used to reliably store and recompute indexes over a slow timescale and a caching layer that is used to respond to most queries. Our primary insight is that the problem of efficiently retrieving a small number of relevant results must be decoupled from the problem of reliably storing potentially huge indexes. Relevant results can be quickly retrieved from caches of ranked results, which can be replicated based upon query load. An entirely different set of mechanisms, such as encoded storage and/or data partitioning, should be used to store large indexes reliably. These indexes should only be used to compute results when there is a miss in the cache.

1. Introduction

Consider the (thought problem) of a completely distributed implementation of Google.[1] In November 2005, Google indexed about 11.2 Billion unique pages (Google no longer publishes the number of indexed pages on its front page; the 11.2B number was inferred by searching for the wildcard "$\star\star$"). The search information is stored in inverted keyword indexes. An inverted index is a list of all objects that match some property, e.g., all documents that include a particular word. With billions of documents, many individual indexes can be very large. Indexes for common words often contain hundreds of millions of entries (369M entries for *Apple*), and indexes for even specialized terms contain hundreds of thousands of entries (437K entries for *NetDB*). Conservatively, each index entry requires 26 bytes (20 bytes for a document ID, and 6 bytes for a IPv4 address, port pair), leading to index sizes ranging from tens of megabytes for specialized terms to tens of gigabytes for popular terms.

What is wrong with Google? Absolutely nothing. The goal of our work is not necessarily to replace (logically) centralized search infrastructures. Instead, our goal is to explore the limits of current decentralized techniques, and present a new architecture which we believe will enable a truly decentralized and distributed search infrastructure. Systems such as Google have demonstrated the immense power of commodity cluster computing coordinated using centralized overview. It is fair to view our work as an attempt at an extension in which we dispense with the centralized control.

Distributing indexes for data sets with billions of documents will require tens (or hundreds) of thousands of hosts. Fortunately, DHTs are particularly good at organizing data over a large number of hosts, and we could conceivably map indexes to hosts using a DHT. This approach has a number of desirable properties. In particular, (1) the reliability of individual hosts is not an issue since the replication in a DHT can cope with usual host failures, and (2) the system can seamlessly scale just by adding new hosts as the corpus increases. This approach, however, has two debilitating problems, one concerned with data storage and the other with result computation. The storage problem is obvious: if the DHT consists of a number of regular hosts, then there is no control over which index(es) gets mapped to which host, and it is likely that the hosts holding the indexes for popular terms will be under-provisioned. The result computation problem is as follows: the usual procedure for computing (conjunctive) queries, consisting of multiple terms, is to intersect the index of each term. Intersecting these indexes requires transferring one or more of the candidate indexes

[1]The authors are not associated with Google; however, they use Google search every day.

over the network. Even if we transfer the smallest index all the time, this procedure consumes a large amount of the system bandwidth. If this system is to serve billions of queries per day, then the network (and processing) overhead of these intersections will prove to be prohibitive.[2] Further, simple intersections do not rank documents by relevance; ideally, the search infrastructure should return only a few documents which are most relevant to the query, and not an unmanageably large set of peripherally related documents. We are not the first to list these problems; a subset of them have been pointed out in [11, 22]. Hence, we assert that in a distributed implementation, (re-)computing results by intersecting complete indexes is not viable. Instead, we make the following assertions:

1. *Results of popular queries must be cached* Cached results allow future queries to be satisfied with bandwidth dependent only on the size of the result, not the size of the indexes used to compute the result.

2. *Results must be ranked* Ranking results allows the most relevant results to be returned to the user without prohibitive network or storage overhead.

3. *Indexes must be partitioned* Indexes are large, and must therefore be partitioned over available hosts in a reliable, fair, and highly available manner.

Motivated by these observations, we propose a two-layer architecture consisting of an index store layer and an explicit caching layer.

The **index store layer** *reliably* stores large indexes, and provides reasonably efficient access to index data. The protocols in the index store layer account for available resources at individual hosts and partition index data accordingly; they also account for expected failure rates and encode/replicate index data such that indexes can be reconstructed when nodes fail. While the store allows complete access to indexes, the protocols are not optimized for frequent accesses due to individual queries.

The **caching layer** provides *efficient* access to popular indexes, and to the results of prior queries. Cached results are useful both for direct hits, and for computing results to new queries that are refinements of previous queries. Cached results are replicated to account for skews in the query distribution, i.e. results for popular queries are cached at more nodes. For almost all queries, users are interested in only the top few *relevant* results, and efficiently fetching the top-k results is an optimization built into many information retrieval tools. For most queries in Google, for example, only the top 1000 unique results are accessible. The ability to store and retrieve a small set of highly ranked results also further reduces the overhead of individual queries.

We believe our two layer decomposition will prove useful because it enables the system designer to decouple reliability and efficiency. This is a key property (and our primary insight) since it allows us to *use well-understood, slower timescale mechanisms for storing and updating large data (the indexes) while still enabling quick access to data that must be accessed quickly (search results).* In particular, the protocols in the index store layer manipulate large data objects and must be reliable: all of the replication here is to ensure that failures do not cause loss of data that is difficult to recreate. Access to the index data need not be instantaneous as long as the caching layer has an adequate hit rate.

The caching layer, on the other hand, is designed for access to relatively small items (top k results for different queries). These items are accessed frequently, and are replicated based on their access frequency. The caching protocols must provide fast access to applicable results. A miss is not fatal, however, since the result can be slowly recomputed using the data stored in the indexing layer.

Will the caching work? Perhaps. Clearly, the most convincing proof would be a billion document high-performance decentralized system, which we unfortunately cannot present in this paper. Instead, our goal for this paper is twofold: first, we hope to convince the reader of the *necessity* of a two-layer search architecture. In the rest of the paper, we describe specific protocols for each layer, discuss their properties and present pointers to related work. Our second goal is to convince the reader that these details constitute sufficient evidence that our approach is interesting, and to motivate future work based upon the ideas in this paper.

2. Two-level Search Infrastructure

Our two-level infrastructure is based on the need to differentiate between large indexes and small result sets, and to store each appropriately. Caching and load-based adaptive replication are clearly crucial techniques, and will be used extensively throughout the system. In the reminder of the section we describe the different components of our architecture.

2.1. The Index Store Layer

The lower level of our two-level architecture is the Index Store Layer, which is charged with storing indexes reliably. We assume that the indexes are generated using an approach similar to the one suggested by Reynolds et al. [16]. Users export their documents when they join the system. For document ranking, a rank vector representing the weight of

[2] The number of queries served is a conservative estimate. While an official count seems difficult to locate, according to an industry article, in February 2003, Google was serving 250M queries per day within the US (http://searchenginewatch.com/reports/article.php/2156461).

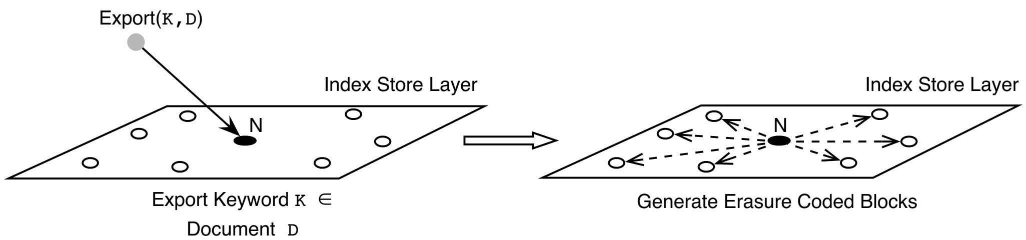

Figure 1. Steps involved in exporting document entries.

each term in the document is generated and the keywords are automatically identified. (See [8] for one approach for generating vectors for distributed ranking.) A document entry (typically consisting of its ID, location and its vector representation) is then added to the index corresponding to keyword. For example, in Figure 1, the index for keyword K is mapped (perhaps using consistent hashing) to node N, and all nodes exporting a document with K will add an entry to the index (logically held) at N. Node N is responsible for reliably aggregating and storing this information.

Partitioning Large Indexes Indexes for popular keywords can grow rather large (tens of gigabytes), and should be stored at multiple nodes due to space and reliability concerns. The simplest approach to solving this problem is to partition the index, and store it at multiple nodes. Different partitioning schemes have been suggested [7, 22], each with different trade-offs between network overhead, balance of data distribution between nodes, and so on. It does not much matter which of these schemes is used in practice, as long as some protocol is used to partition the indexes.

Redundant Storage Node churn and failure is a frequent event in P2P systems. If we are to build a search infrastructure over unreliable peers, critical data such as indexes must be replicated. Two mechanisms are viable: keeping multiple copies or using coding for redundant storage. Both have been investigated in recent literature: DHT designs often replicate data at a fixed number of nodes determined by the name-to-key mapping. Rodrigues et al. [17] derive a relationship between the number of replicas, the failure probability and the probability of restoring the data item. Their results indicate that in order to recover the data with 0.9999 probability, we would need 4 replicas when 10% of the nodes fail and 14 replicas when 50% of the nodes fail (failing nodes chosen uniformly at random in each case).

Research by Weatherspoon et al. [24] and Rodrigues et al. [17], however, shows that erasure codes (such as Reed-Solomon [15] codes) are superior to replication in terms of space and bandwidth requirements, especially when the environment is extremely dynamic. Based on the derivation in [17], encoding the index for *Apple* only requires 1.72 times more space than the single copy of the index when 10% of the nodes fail, and 5.18 times the space with 50% failures. Note that coded storage comes with an extra cost: updates to data requires encoded blocks to be reassembled and then recoded, as opposed to just appending to an existing replica.

In practice, we envision a hybrid scheme in which one copy of the index is stored (partitioned among different nodes) without encoding and replicas are stored using erasure-coded blocks. Such a design (modulo the partitioning of the large indexes) has previously been proposed by Rodrigues [17]. Here, when a node holding a partition fails or departs, the partition is re-created using appropriate erasure-coded blocks. Recall that if erasure codes are used, the entire set of codes need to be re-created for each update. This overhead can be minimized by batching updates for each partition and publishing the new set of encoded blocks periodically.

Computing Relevant Results The final operation required of the Index Store is to compute (ideally once) the relevant results of a query. Different methods can be used to rank results; we have demonstrated one technique in [8] that uses Vector Space Models [19]. Since information about all possible query results are present within the Index Store layer, there is sufficient information here to rank order query results and extract only the top few results.

2.2. The Caching layer

We have asserted that result caching is *necessary* for a large scale search infrastructure. This is because the overhead of recomputing results by intersecting indexes is simply prohibitive. For example, even a specialized query such as "NetDB 06" would intersect indexes with 434K and 711M entries. A smart implementation would transfer the smaller index (434K entries, about 10MB of data) over the network, and this could further be reduced to 425KB using Bloom filters (See [16]), or even lower using Compressed Bloom filters [13]. However, if Bloom filters are used, then the larger index (with 711M entries) would have to be scanned, leading to a high processing cost per query.

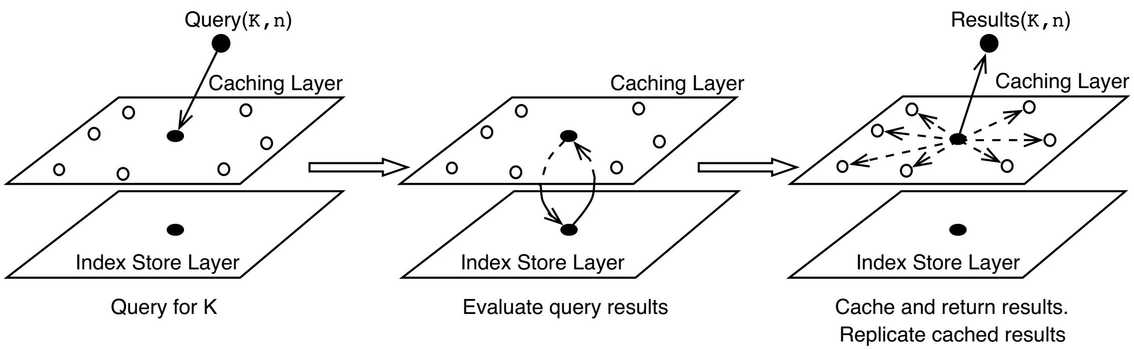

Figure 2. Steps involved in query evaluation and result caching.

Thus, the overhead, either in terms of bandwidth (if Bloom filters are not used) or processing (if Bloom filters are used), especially multiplied over billions of queries, is prohibitive.

The different steps involved in result caching are shown in Figure 2. All queries are initially directed to the caching layer. The query terms are used to locate appropriate caches, and if the query can be satisfied within the caching layer, results are immediately returned. Upon a miss, the caching layer transparently locates the appropriate index in the Index Store layer and requests the requisite number of top results for the query. The Index Store layer computes the relevant results using the procedure described in Section 2.1, and returns the results. Finally, nodes in the caching layer replicate popular results depending on the query load directed towards individual caches.

Global Result Caching The challenge with caching results lies in storing the caches such that they can be efficiently located and re-used throughout the system. We have designed a distributed data structure called the View Tree [1] to facilitate the storage and location of these caches. Each node in the View Tree represents the cached results of a conjunctive query. The View Tree also enables the use of cached sub-queries while evaluating queries. The problem of identifying the smallest set of sub-queries that can be used to evaluate the conjunctive query, however, is NP-hard (by reduction from Exact Set Cover [5]). Hence, we use a set of heuristics to identify the caches useful in computing the results of a query. Our results show that View Trees can reduce the amount of data transferred for query evaluations by more than 90%.

Handling Hot-Spots Differences in the popularity of keywords results in nodes hosting popular indexes receiving more load compared to the other nodes. While one could tackle the problem by creating sufficient replicas, it is hard to predict the query load of an index *a priori*, and the popularity of keywords changes over time. This makes static replication of indexes inefficient. However, several load-based dynamic replication techniques are known, including protocols that change replica locations based on access patterns [25], protocols that replicate on the query source–destination path [4], our own Load Adaptive Replication protocol tailored for P2P environments [9], protocols tailored for power-law query distributions [14], and randomized load-adaptive replication protocols [2].

3. Related Work

Li et al. [11] question the feasibility of Web indexing and searching in P2P systems. Their analysis shows that existing techniques for searching require more bandwidth than is available. Our two-level structure is motivated exactly by these observations, and we believe the techniques proposed in this paper make distributed search viable. Our architecture provides scope for innovation in a large number of areas, including lookup, storage, result computation, caching, etc. In the rest of this section, we present an overview of prior work in these areas.

Lookup Our architecture builds on prior work on efficient lookup and storage schemes. We assume the existence of a lookup protocol provided by the underlying P2P system. Such lookup protocols have been studied in detail in Chord [20], Pastry [18], Kademlia [12], Skipnet [10], etc.

Storage and Partitioning Weatherspoon et al. [24] and Rodrigues et al. [17] study the trade-offs between erasure codes and replication. Weatherspoon et al. show that, for the same levels of reliability, erasure codes require much less space compared to replication. Rodrigues et al. derive the relation between reliability, probability of failure and the number of replicas or erasure coded blocks. They show that the cost of using erasure codes is similar to that of replication when the failure rate is low, but yields substantial savings under high failure rates. Tang et al. [22] and

Gopalakrishnan et al. [7] study different approaches to partitioning index data. The difference in the two approaches lies in the amount of network traffic and the load balance resulting due to the partitioning.

Result Computation and Caching Building a search facility over P2P systems has been an important area of research. Reynolds et al. [16] were the first to propose the use of inverted indexes for searching and the use of Bloom filters for computing intersections. Odissea [21] shares our vision of having a distributed infrastructure to perform full-text searches using inverted indexes. Peers are arranged in a Chord-like ring and store indexes that get mapped to them. Odissea, however, does not address many of the practical problems associated with P2P systems such as index sizes, query load and bandwidth limitations. eSearch [22] is a boolean query system for textual data over Chord. In eSearch, indexes store all the keywords in the document along with the document entry; this eliminates the need to contact the indexes of all the keywords in a conjunctive query. eSearch, however, does not cache query results nor does it have the ability to rank the results. Gnawali [6], in his thesis, proposes storing indexes that are intersections of two keywords. The design is motivated by the fact that a large fraction of the queries contain two or more keywords. Li et al. [11] discuss the use of client-side caching to reduce the amount of data transferred. They show that caching results reduced the communication cost by 38%. We proposed [1] the use of view trees for caching results and showed that we can reduce the communication cost of evaluating queries by about 90%.

Similarity Search and Ranking pSearch [23] supports similarity-searching by mapping document vectors to a high-dimension P2P system. The query is also mapped to the same space and controlled flooding is employed to fetch relevant documents. Bhattacharya et.al. [2] use similarity-preserving hashes (SPH) to extend pSearch to any DHT-based system. Ranking results in distributed search is an exciting new area: we have presented [8] a distributed VSM-based approach to rank query results. PlanetP [3] is a content-based search scheme that uses gossiping to spread the meta-data of the content stored at each node. To evaluate results, *peers* are selected by ranking them using the gossiped information and the query is then evaluated using VSM in these select peers.

4. Discussion and Open questions

We believe that our two-level decomposition conceptually removes the most critical barriers from the realization of a wide-area distributed search infrastructure: that of reliably maintaining and accessing large indexes, and that of efficient result computation. However, there remain many open questions, which we classify in three broad areas:

Index Storage There is much work still needed in understanding how to store large objects in decentralized systems, including DHTs. While data partitioning is a prerequisite, the amount of data that must be moved when a node joins/leave the system may render the entire system unusable. The overall impact of the replication versus erasure coding choice is not clear. While erasure codes have advantages in bandwidth and storage overhead, they require contacting many more nodes than does replication. Further, a deterministic scheme is needed to publish erasure coded blocks in such a way that they can be easily retrieved, but the best method to do this is not yet obvious. Batching can reduce the overhead of using erasure codes. However, it also increases the chance of losing updates due to failures. Finally, the effect of the update period on overhead and reliability needs to be analyzed.

Query Distribution, Caching and Ranking The characteristics of Google-like data sets and query streams need to be tightly characterized. Highly dynamic systems and variable access patterns can play havoc with complicated caches. If the most relevant documents for a given query change, the cache must be updated or the quality of the results degrade. Clearly this issue is highly dependent on the characteristics of the data sets and query streams used, but traces of systems like Google are prohibitively large. We would ideally like to create a set of relatively small traces that are provably representative of Google-like systems. Failing this, there is still a great deal of research that can be done studying properties of systems like the one proposed in this paper, such as analyzing how quickly cached results degrade with increasingly dynamic systems.

Security and System Model An unstated and somewhat naive assumption in this paper has been that of an entirely cooperative (and non-malicious) set of peers, and we have designed the system to withstand random node failures only. While to the best of our knowledge our architecture does not enable any *new* security holes, there are any number of attacks that can render the system essentially useless. A number of these attacks require formulation of global policy, e.g. how to handle nodes that publish junk data to fill up all available space? Other attacks include nodes that selectively deny service to other nodes, or respond with spurious results. We believe the encoded storage techniques can help the resilience of the system since they allow data to be reconstructed even if a large number of nodes are malicious/attacked; however, a systematic study of their resilience to different attacks is open. Spurious data supplied by malicious nodes is not an issue if all data is selfcertifying, but in a wide-area network, it is not clear how third-party signatures can be checked without a trusted CA or a PKI. If a decentralized system is to be widely used for search, all of these issues must eventually be addressed explicitly.

References

[1] B. Bhattacharjee, S. Chawathe, V. Gopalakrishnan, P. Keleher, and B. Silaghi. Efficient peer-to-peer searches using result caching. In *Proceedings of the* 2^{nd} *International workshop on Peer-To-Peer Systems*, Berkeley, CA, March 2003.

[2] I. Bhattacharya, S. R. Kashyap, and S. Parthasarathy. Similarity searching in peer-to-peer databases. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 329–338, 2005.

[3] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. In *Twelfth IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*. IEEE Press, June 2003.

[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 Systems Principles*, Banff, Canada, October 2001.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

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

[7] V. Gopalakrishnan, B. Bhattacharjee, S. Chawathe, and P. Keleher. Efficient peer-to-peer namespace searches. Technical Report CS-TR-4568, University of Maryland, College Park, MD, February 2004.

[8] V. Gopalakrishnan, R. Morselli, B. Bhattacharjee, P. Keleher, and A. Srinivasan. Ranking search results in peer-to-peer systems. Technical Report CS-TR-4779, University of Maryland, College Park, MD, January 2006.

[9] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive replication in peer-to-peer systems. In *The 24th International Conference on Distributed Computing Systems*, Tokyo, Japan, March 2004.

[10] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Seattle, WA, March 2003.

[11] J. Li, B. T. Loo, J. Hellerstein, F. Kaashoek, D. Karger, and R. Morris. On the feasibility of peer-to-peer web indexing and search. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, Berkeley, CA, Feb 2003.

[12] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *First International Workshop on Peer-to-Peer Systems (IPTPS '01)*, pages 53–65, London, UK, 2002.

[13] M. Mitzenmacher. Compressed bloom filters. In *20th ACM Symposium on Principles of Distributed Computing (PODC '01)*, pages 144–150. ACM Press, 2001.

[14] V. Ramasubramanian and E. G. Sirer. Beehive: Exploiting power law query distributions for O(1) lookup performance in peer to peer overlays. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, pages 331–342, San Francisco, CA, USA, March 2004.

[15] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *J. Soc. Indust. Appl. Math*, 8:300–304, 1960.

[16] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of IFIP/ACM Middleware 2003*, 2003.

[17] R. Rodrigues and B. Liskov. High availability in DHTs: Erasure coding vs. replication. In *Proceedings of the 4th International Workshop on Peer-to-Peer systems (IPTPS)*, February 2005.

[18] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM Middleware 2001*, Heidelberg, Germany, November 2001.

[19] G. Salton, A. Wong, and C. Yang. A vector space model for information retrieval. *Journal for the American Society for Information Retrieval*, 18(11):613–620, 1975.

[20] 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.

[21] T. Suel, C. Mathur, J. Wu, J. Zhang, A. Delis, M. Kharrazi, X. Long, and K. Shanmugasundaram. Odissea: A peer-to-peer architecture for scalable web search and information retrieval. In *6th International Workshop on the Web and Databases (WebDB)*, June 2003.

[22] C. Tang and S. Dwarkadas. Hybrid global-local indexing for efficient peer-to-peer information retrieval. In *Proceedings of USENIX NSDI '04 Conference*, San Fransisco, CA, March 2004.

[23] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *Proceedings of ACM SIGCOMM '03 Conference*, pages 175–186, Karlsruhe, Germany, 2003. ACM Press.

[24] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proceedings of the 1st International Workshop on Peer-to-Peer systems (IPTPS)*, March 2002.

[25] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, 1997.