Distributed Web Crawling over DHTs

Boon Thau Loo Owen Cooper Sailesh Krishnamurthy
{boonloo, owenc, sailesh} @cs.berkeley.edu
University of California, Berkeley

Abstract

In this paper, we present the design and implementation of a distributed web crawler. We begin by motivating the need for such a crawler, as a basic building block for decentralized web search applications. The distributed crawler harnesses the excess bandwidth and computing resources of clients to crawl the web. Nodes participating in the crawl use a Distributed Hash Table (DHT) to coordinate and distribute work. We study different crawl distribution strategies and investigate the trade-offs in communication overheads, crawl throughput, balancing load on the crawlers as well as crawl targets, and the ability to exploit network proximity. We present an implementation of the distributed crawler using PIER, a relational query processor that runs over the Bamboo DHT, and compare different crawl strategies on PlanetLab querying live web sources.

1 Introduction

Search engines such as Google [2] have become an integral part of the Internet. These systems typically expose a limited search interface to end-users through which users enter search terms and receive results according to the engine's ranking function.

Beneath this interface, search engines are generally comprised of three core components. First, the *crawl* component trawls the web and downloads web content to be cached locally. Second, the *index* component precomputes indexes for the cached web content for efficient search. Last, the *search* component uses the indexes for executing user search queries returning ranked results.

This interface has served us well when the web consisted of a relatively small set of fairly static web pages. As the web has grown and evolved towards dynamic content, however, search engines face increasing challenges in maintaining a fresh index over the entire web. As a result, search engines today index only a fraction of the web, and design their crawlers to prioritize updates of some web sites over others. It is estimated that Google indexes around 3 billion web documents, which is a tiny fraction of the 550 billion documents [9] mostly within the *deep web* estimated in the year 2001.

Further, search engines are known to censor and skew results rankings [7], mostly in response to external pressures. Examples of Google's censorships abound in response to DMCA complaints from KaZaA [4], The Church of Scientology's attempts to silence its critics, and the banning of seditious web sites by certain countries. If Google is the guardian of the web, it is reasonable to ask: *quis custodiet ipsos custodes ?*[1].

To address the shortcomings of centralized search engines, there have been several proposals [16, 25] to build decentralized search engines over peer-to-peer (P2P) networks. In this paper, we focus on the design issues of *distributed crawling*, which is an essential substrate for any of the proposed decentralized search applications. The web content harvested by a distributed crawler can be indexed by decentralized search infrastructures, or archived using a persistent storage infrastructure such as OceanStore[15]. Here, we focus our discussion on crawling and do not address the orthogonal issues of persistent storage and indexing.

The distributed crawler harnesses the excess bandwidth and computing resources of clients to crawl the web. At a high-level, crawls are expressed as *recursive queries* and executed using PIER [14], a P2P relational query processor over DHTs. A DHT provides useful properties of load balancing, reliable content-based routing in the absence of network failures, and logarithmic (in the number of nodes) lookup costs. This allows us to easily dispatch crawl requests evenly across crawler nodes in a decentralized fashion.

The query is executed in a distributed fashion as follows. Each *crawler node* runs the PIER query engine and is responsible for crawling a different set of web pages. The query is first sent to all the crawler nodes, and set to run according to some exit criterion (crawling time, depth of crawl etc.). A URL is partitioned amongst the participating crawler nodes by publishing it in the DHT. The partitioning (discussed in Section refsec:crawl) scheme) is determined by how URLs are published into the DHT such as hashing the entire URL or only its hostname. Each node is responsible for crawling the URLs published in its partition of the DHT. The crawl is begun by publishing a set of seed URLs starting a distributed, recursive computation in which the PIER query continuously scans the DHT for new URLs to be crawled. A web page is downloaded for each URL crawled the links it contains are refined according to user predicates and then republished into the DHT for further crawling.

Note that the distributed query processor naturally coordinates and partitions the work across the participating nodes. Using only data partitioning of the intermediate URLs to be

[1]Who will guard the guards ?

processed, it parallelizes the crawl without explicit code to ensure that multiple sites do not crawl the same web pages redundantly.

The potential of such a distributed crawler is immense. We envision this distributed crawler to be used in the following ways:

- **Infrastructure for Crawl Personalization:** Users can customize their own crawl queries and execute them using this service. Special interest groups can also perform *collaborative filtering* by executing a query that matches the group's interests.
- **High-throughput Crawling:** By harnessing the power of a large number of nodes, the crawling service is more scalable than centralized systems.
- **Generalized Crawler for P2P Overlays:** Although we focus our attention on web crawling, our ideas can be used to build a generalized crawler for querying distributed graph structures over the Internet [17]. Such a generalized crawler can be used to run queries over the link structure or meta-data of P2P networks such as Gnutella [1] and OceanStore [15].

The rest of this paper is organized as follows. Section 2 provides a high-level overview of the distributed crawler. In Section 3, we describe a detailed description of the crawl query execution of our distributed crawler. Next, in Section 4 we present a comparison of different query execution strategies by running our crawling service on nodes of PlanetLab [5] crawling live web sources. We survey centralized and parallel crawlers as well as other distributed web crawling schemes in Section 5. Finally, we present an agenda for future research in Section 6 and summarize our conclusions in Section 7.

2 System Overview

In this section, we present an overview of the distributed crawler and describe the goals that guide our design.

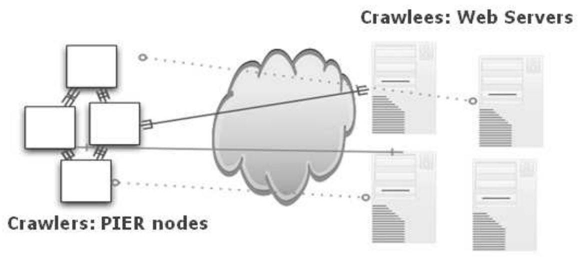


Figure 1: *Distributed Crawler using P2P nodes*

The distributed crawler is designed to harness the excess bandwidth and computation resources of nodes in the network to crawl the web. Figure 1 shows the deployment model, where *crawler* nodes on the left collectively pool their resources together to crawl the *crawlees*, that are the web servers on the right. This model is similar to that of SETI@Home [6] and Grub [3], both of which pool resources of participating nodes in a network for common computation. Our system is distinguished by the lack of a central coordinator to distribute work, and the fact that crawl queries can be issued from any node in the system.

2.1 Design Goals

We used the following goals while designing our distributed crawler:

- **Ease of User Customization:** Our crawler is designed to be easily customizable. Crawls are specified as *recursive queries* [18] whose results can be defined in terms of the query itself. Recursive queries are particularly useful to query recursive relations such as the reachability of a network graph. In our system, users can specify where the crawl begins, define *predicates* that filter the pages to be crawled, and also control the order in which these pages are crawled.
- **Ease of Composability:** We designed the distributed crawler entirely out of *off-the-shelf* components from existing research projects. We use the Telegraph Screen Scraper (TeSS) [8] to download and parse web pages. Crawl queries are executed using PIER [14], a P2P relational query processor over DHTs. While PIER can use different DHTs, we use Bamboo [20] DHT in our implementation.
- **Query Processor Agnostic:** Currently, PIER provides us with an interface to express queries in "boxes-and-arrows" dataflows. In future, we will provide an additional declarative interface that will generate these dataflow diagrams given the declarative recursive queries. The use of declarative queries to express crawls will enable us be agnostic to the query-processor, hence allowing the crawl query to be executed on any centralized, parallel or distributed query processor that provides support for recursive queries. For example, a similar crawl query executed using PIER can also be executed as a centralized crawler using Telegraph [11].
- **Scalability:** As the number of crawler nodes increases, the *throughput* (number of bytes downloaded per second) of the crawlers should grow. The throughput can be maximized by two techniques: (1) load on crawler nodes must be balanced so that they are not idle and (2) reducing the DHT communication while executing the crawl.
- **Being good netizens:** The crawler can be easily abused to launch distributed denial of service (DDoS) attacks against web servers. We will provide mechanisms to limit (throttle) the rate at which web sites are accessed.

There are other practical issues such as fault tolerance that are important in order for the system to be deployed. These are beyond the scope of this paper and are outlined as part of future work in Section 6.

3 Crawl Query Execution

In this section we describe how crawl queries are executed in our system. We first describe the crawler using a simple *partition by URL* scheme to distribute the crawl. Subsequently, we will discuss other distribution schemes.

Figure 2 shows a simple crawl query execution plan. When executed, this query plan starts a web crawl from a set of seed URLs, extracts links from the web pages downloaded, and publishes the links back to the DHT. Later, we will show how this simple query can be modified to construct inverted indexes of the web pages over the DHT. It is also easy to modify the query to archive the web pages themselves in a persistent storage infrastructure like OceanStore.

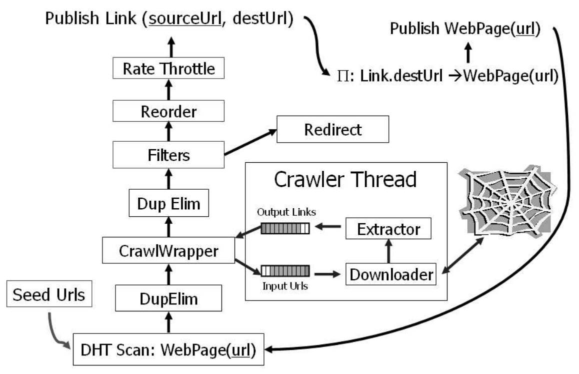


Figure 2: *Expressing Crawl as a Query*

The query execution plan is expressed as a "boxes-and-arrows" dataflow, supporting the simultaneous execution of multiple operators that can be pipelined to form traditional query plans. The query is recursive, as shown by the presence of a cycle in the plan. A declarative form of the query can be expressed in Datalog [18]. The query can be terminated either based on the duration or depth of the crawl or when there are no more new URLs to be crawled.

This query plan is sent to all the crawler nodes on initialization. The crawl begins by partitioning a set of initial seed URLs. This is done by publishing `WebPage(URL)` tuples into the DHT, where the underlined field is used to hash the tuple into the DHT. Each crawler node continuously scans their local DHT partition for new `WebPage` tuples. When a new tuple arrives, the scan operator passes it up the query plan through a *DupElim* operator where duplicates are eliminated. This avoids redundant work as the same URL may be discovered separately by different crawlers. The `WebPage` tuple is then sent to a special *CrawlWrapper* operator that uses input and output queues to interface with *crawler threads* that are workhorses for screen scraping. The crawler threads use a generic TeSS wrapper to download and parse web pages.

Each `WebPage` tuple is added to the input queue of URLs in one of the crawler threads. A crawler thread dequeues each URL, downloads and parses web pages, extracts links to generate `Link(sourceURL, destinationURL)` tuples that it enqueues in its output queue. The crawler uses an *extractor* module that uses regular expressions to identify link anchors. Once extracted, these URLs are post processed to resolve them to an absolute URL. Finally, these URLs are paired with the page from which they originated to form `Link` tuples.

The CrawlWrapper dequeues links from the crawler thread and sends them to another *DupElim* operator for further duplicate elimination. Notice that we need the second DupElim as the generated links might have already been seen before. Each new link is then filtered with special user-defined predicates, to decide if it should be followed. The filtered links can be prioritized with the *Reorder* operator and are periodically published back to the DHT at a rate determined by the *Rate Throttle* operator. Each `Link` tuple is published back into the DHT using its `sourceURL`. Note that is the same URL that was used for downloading, and so is a local put in the DHT incurring no network overheads. The `destinationURL` field is then projected to form a new `WebPage(URL)` tuple, which is then published back into the DHT, for further crawling on another crawler node.

3.1 Crawl Enhancements

Currently, apart from setting the seed URLs, crawls can be customized by specifying user-defined filters and *Reorder* operator. We have implemented two simple filters: one that limits the crawl based on its depth, and another based on matching a substring of the `destinationURL` field of `Link` tuples. For example, to limit the crawl within the Google.com domain, we can perform substring matching on `destinationURL` field to only follow links containing the term `google.com`.

Users can also customize the *Reorder* operator that decides the order in which newly discovered links are crawled. One simple scheme is for the extractor to compute a priority for each link, based on the relevance of the downloaded content, and then pass that as an extra field in the `Link` tuple for the *Reorder* operator to make decisions. Apart from reordering, rate throttling can also customized on a per-server basis.

The simple link extraction query can be easily extended to support more complicated queries. For example, the *extractor* extract keywords from the downloaded web pages, which can then be used to build inverted indexes in a DHT over the web pages. This allows for DHT-based keyword searching over the crawled web pages.

3.2 Crawl Partition Schemes

The query described above uses a *partition by URL* scheme to distribute the crawl workload. While this scheme balances load well amongst the crawler nodes, it will likely incur high communication overheads in publishing `WebPage` tuples.

The alternative to this scheme is to *partition by URL's hostname*, effectively dedicating a crawler node for each unique web server. This has the advantage of not incurring any DHT communication in publishing `WebPage` tuples for *self-links*. To implement this, we added a *partitionURL* field to each `WebPage` tuple, set it to be the hostname of the URL, and use it as the hashing key while publishing `WebPage` tuples in the DHT.

In addition to lowering DHT communication, this scheme has a single control point for access to each web server, providing an easy way to perform per-server rate-throttling without having to coordinate a possibly large set of crawler nodes. Despite its benefits, this may lead to poor balancing of crawling load as some nodes may be responsible for crawling web sites that serve a large number of pages. It also introduces a single point of failure, where a "bad" choice of node may severely affect per-server crawl throughput. A natural extension of this scheme is to assign n crawler nodes to each web server. We call this scheme *partition by hostname n*. Currently, n is determined statically, and in future, we will explore techniques for varying n dynamically.

3.3 Redirection

Redirection provides a simple technique for allowing a crawler to pass on its assigned work to another crawler. Redirection works with any of the partitioning schemes described above, and is a *second chance* mechanism that allows a page request to be sent to a different crawler node. Redirection is performed simply by having the crawler thread *bounce back* the original `WebPage` tuple that is sent to the *Redirect* operator for publishing in the DHT for further processing.

The redirection scheme can implement a number of optimizations. For instance, a node might choose to redirect a `WebPage` tuple to another crawler node if it has high network latency from the target web site. Redirection is also useful both for distributing load amongst crawlers and to limit the rate at which web sites are crawled. To illustrate, in the *partition by hostname* scheme, a node with responsibility for a high traffic site may initially decide to crawl all links from the site. Subsequently, when overloaded, the node can become a *redirector node*, and redirect traffic to other nodes. This achieves better load-balancing, and retains the single control point property of the redirector node for easy rate throttling.

Apart from simply redirecting by URL, we can also control the fanout of redirection. For example, to redirect a request to n different nodes, we can simply create a *partitionURL* field in the redirection tuple that consists of the original *partitionURL* appended with a random number in $[0, n)$.

A big drawback of redirection is the extra DHT communication overheads incurred. To alleviate this, we set a threshold on the number of redirections each `WebPage` tuple can have. Implementing this threshold requires an extra `redirectCount` field to be stored in `WebPage`.

4 Evaluation

In this section we report the results of an experimental study of our distributed crawling service. The crawler was deployed on up to 80 nodes of PlanetLab. We execute the simple link extraction query described in section 3, and examined four crawl distribution strategies under three different workloads on live web sources. The number of crawler threads on each node was limited to three, both to limit the load on PlanetLab and also to simulate the resources that a typical crawler node owned by a home user might have to spare. As a result of the small number of crawler threads, the throughput of the crawler tends to be affected by the number of active crawler threads. For each separate experiment we varied the number of participating crawlers from 1 through 80. In order to take into account differing loads on PlanetLab, we averaged our experiments across as many as five runs.

4.1 Experimental Setup

We now list the three crawl workloads we studied:

- **(A) Exhaustive Crawl:** In this setup, we started the crawl with the seed URL `http://www.google.com`. For each web page downloaded, *every* hyperlink in the page is distributed for further crawling. In all, requests are made for more than 800000 pages from over 70000 different web servers.

- **(B) Crawl of multiple fixed web sites:** Here, we crawl a static fixed set of web servers. Again, we start the crawl with `http://www.google.com`, but only distribute URLs served off the Google domain for further crawling. An example of such a server is `http://www.google.com.au`. In all, requests are made for more than 500000 pages from 45 web servers. This simulates the workload of performing focused crawls on a small set of pages.

- **(C) Crawl of single web source:** In this setup, we constrain the crawl within `http://groups.google.com`, harvesting as many newsgroup articles as possible over the duration of the crawl.

We compared four different partition schemes: (1) *partition by URL*, (2) *partition by hostname* (3) *partition by hostname-8* across eight crawlers and (4) a one-hop *redirection* scheme which behaves like *partition by hostname* switching to *partition by URL* when the crawler's pending input queue size exceeds 500. In our graphs, and in the rest of this section, they are referred to as *URL*, *Hostname*, *Hostname-8* and *Redirect* respectively.

4.1.1 Limitations

In our experiments, all crawls lasted for a period of 15 minutes. Apart from filtering the URLs based on their domain, there were no content-based filters to prune exploration of newly discovered web pages. The crawl generates an ever-increasing number of new links to be explored. Hence, the number of page download requests for each crawler node exceeds the actual number of downloads. For example, in one exhaustive crawl experiment on 80 crawlers, 873631 page requests were made, but only 43404 were actually downloaded by the crawlers. In practice, this problem can be alleviated by limiting the crawls to a finite set of pages. We leave implementation of content-based filters for future exploration.

4.1.2 Metrics

We study the performance of our crawling service based on the following experimental metrics:

- **Crawling Load:** The actual crawling load on each participating node is measured in terms of the total number of bytes downloaded over the crawl period and is expressed in kilobytes per second (KBps). To evaluate how well the load is balanced, we examine only the case where all 80 nodes participate in a crawl. We consider the *cumulative distribution function* (CDF) of the crawling load to measure how it is distributed amongst all nodes.

- **DHT Load:** The DHT load for each node is the incoming traffic due to the DHT. PIER collects this statistics as part of its normal operation,, and is expressed in kilobytes per second (KBps). As in the case with crawler load, we consider the CDF of the DHT load over all nodes for the case where all 80 nodes participate.
- **Throughput:** The throughput for a given configuration is the total crawling load accumulated over *all* participating nodes and is measured in kilobytes of web pages downloaded per second.
- **Communication Overheads:** The communication overheads is the total DHT load accumulated over *all* participating nodes during query execution. Since different experiments result in a differing number of pages crawled, we *normalize* the communication overhead by considering the ratio of the accumulated DHT load over the course of the query and the number of distinct page requests made. The communication overheads do not include the cost of initial query dissemination, and for long running queries, is expected to be dominated by the cost of rehashing `WebPage` tuples.

Although one might expect that the crawling and DHT loads at each node are correlated, this may not be true during redirection, because the decision to redirect incurs communication overheads even if the actual download occurs on a different node. The throughput for a given configuration is mostly a function of the number of active crawler threads during the course of the experiment, and is directly affected by how well the load is balanced across the crawling service.

4.2 Exhaustive Crawl

Here we consider workload (A) which performs an exhaustive crawl starting from `http://www.google.com`.

4.2.1 Crawling and DHT Load

In Figure 3, we show how the crawling and DHT load is balanced over 80 participating nodes. When the crawling load is perfectly uniformly partitioned over the DHT, each node is responsible for an equal portion of the ID space and all nodes incur the same bandwidth usage. In this case we see a vertical line[2] for both the crawling and DHT loads. In our experiments, however, some nodes are responsible for a larger fraction of URLs than others. As a result, even for the *URL* and *Redirect* schemes where we can expect even load distribution, not all nodes have received the same crawl load. These two schemes still had a more balanced load distribution across the 80 nodes compared to the *Hostname* scheme.

[2]We verified this for the *URL* scheme in simulation where each node was assigned an equal portion of the ID space

As the exhaustive crawl covers a large variety of hosts, even *Hostname* shows a good distribution of load across nodes. However, some load imbalance is unavoidable for *Hostname* even with a large variety of hosts. The busiest nodes of the *Hostname* scheme incurred a crawling load of roughly 43KBps compared to only 25KBps for those of the *URL* scheme. Partitioning each hostname across 8 crawlers alleviates the load on the busiest node for *Hostname*, lowering the peak bandwidth from 43KBps to 27KBps.

In terms of DHT load, *Hostname* is the lowest. The DHT load is again not balanced perfectly because of non-uniform partitioning in practice.

4.2.2 Throughput

Figure 3(a) shows that none of the schemes have *idle* nodes that are not involved in the crawl process. This implies that for most of the time, crawler threads are busy. Hence, the throughput is similar for all the schemes as shown in Figure 4. The throughput scales linearly as the number of participating nodes increase.

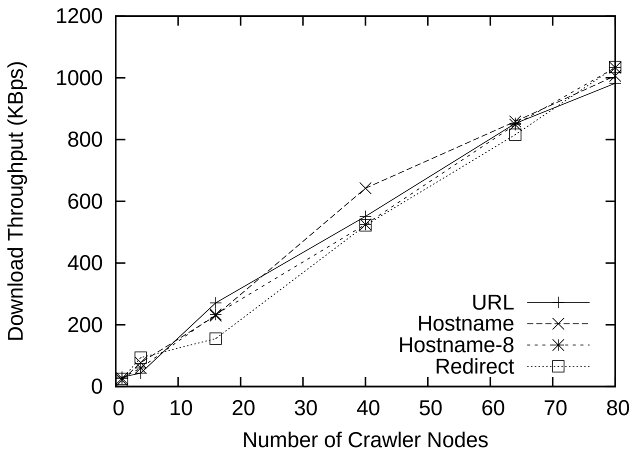
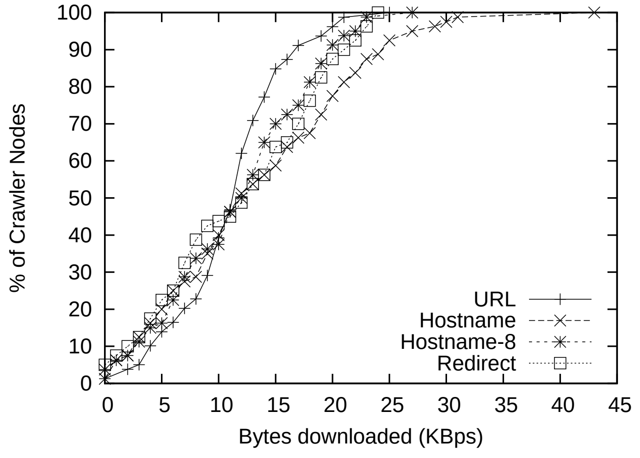


Figure 4: *Throughput (Exhaustive Crawl)*

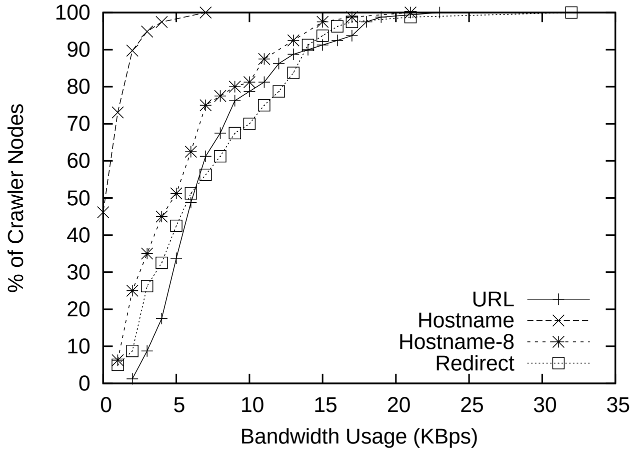
4.2.3 Communication Overheads

Figure 5 shows the normalized DHT load varying over participating crawler nodes. We make the following observations:

1. The normalized bandwidth starts from 0 (no DHT load) when there is only 1 node, and increases sharply as the number of nodes increases. However, it levels off once there are 16 participating nodes. This is because as the number of nodes increases, each `WebPage` tuple that is published as the control message becomes a remote operation with high probability. So with n nodes, the probability of a remote operation for the URL scheme is $(n-1)/n$, which quickly converges to unity as n increases.
2. *Hostname* has the least overheads, followed by *Hostname-8*, *URL* and *Redirect*. Hostname incurs the least overhead because self-links result in zero repartitioning with only local DHT puts. In *Hostname-8*,



(a) CDF of crawling load (bytes downloaded)



(b) CDF of DHT load (DHT traffi c)

Figure 3: Exhaustive Crawl: Crawling and DHT loads balanced over 80 nodes

each self-link has a $7/8$ probability of generating non-local DHT puts, which is less than the probability of $(n-1)/n$ for *URL* when n exceeds 8. So it performs slightly better than *URL* in terms of bandwidth usage.

3. *Redirect* incurs the highest overheads despite using *Hostname*, as redirected requests use the *URL* method when crawler queue sizes exceeds 500. Redirection requires extra fields that inflates the size of each `WebPage` tuple by 50%. With this scheme suffering from larger `WebPage` tuple sizes, *Redirect* requires the most DHT bandwidth.

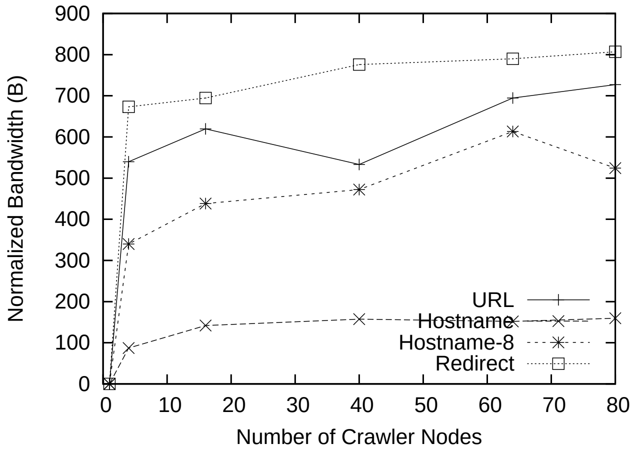


Figure 5: *Normalized DHT Load (Exhaustive Crawl)*

4.3 Crawl of Multiple Fixed Web Sites

We next consider workload (B) which starts with the `http://www.google.com` seed URL but restricts the crawl to 45 web servers in the Google domain.

4.3.1 Crawling and DHT Load

Since *Hostname* uses at most 45 crawlers, Figure 6(a) shows that 70% of the crawlers are idle for *Hostname*. Even with the use of *Hostname-8* there are still 20% of idle nodes. On the other hand, both *URL* and *Redirect* are load-balanced, the latter scheme switching quickly from *Hostname* to *URL* when the queue sizes are exceeded. *Redirect* has 1% idle nodes while there are none for *URL*.

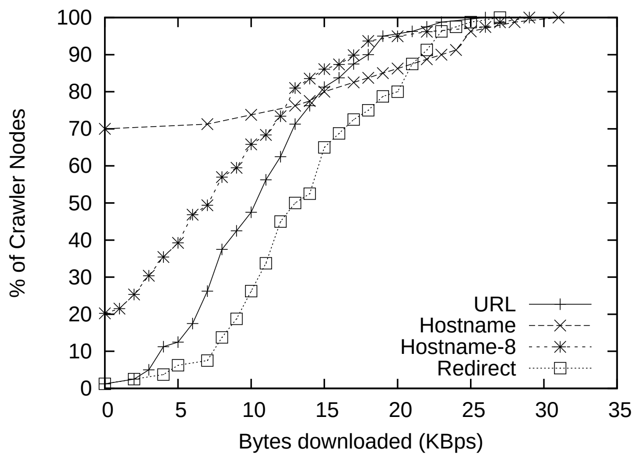
Figure 6(b) also shows that *Redirect* leads to DHT load imbalance. 60% of the nodes incur the same overheads as *URL*, but the remaining 40% incur much higher DHT load as a result of being redirection nodes. This imbalance is caused by redirection, where nodes that have to redirect requests incur higher DHT load compared to those that do not. *Hostname* and *Hostname-8* show even worse imbalance, with a large fraction of nodes incurring less than 1KBps DHT load.

4.3.2 Throughput

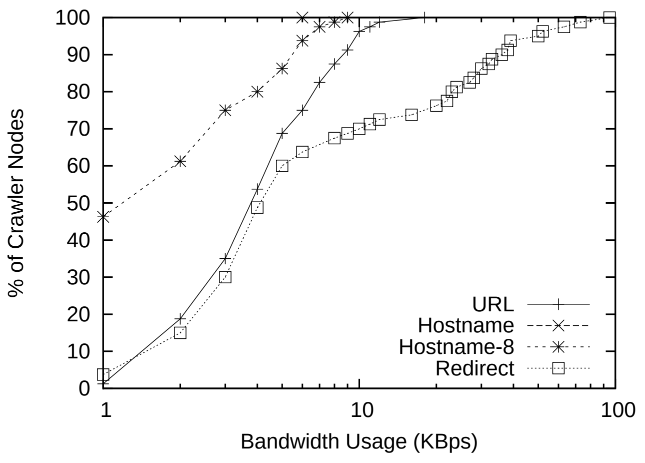
Figure 7 shows that the crawler throughput is greatly affected by the partition scheme used since the fraction of idle nodes differ greatly across the different schemes. *Hostname* has the worse throughput as it has the largest number of idle threads. But *Redirect* and *URL* display good throughput due to the low fraction of idle nodes. *Redirect* shows a 20% throughput improvements over *URL*, hence showing evidence that a load-based redirection scheme can lead to improved throughput in this workload.

4.3.3 Communication Overheads

Despite the benefits of *Redirect*, figure 8 shows that *Redirect* can incur up to 3 times as much DHT load compared to *URL*. Under this workload, 90% of page requests are redirected, hence leading to the high overheads. In fact, in the steady state, the queue sizes in the redirector nodes are exceeded quickly, and each page request requires two rehashes,



(a) CDF of crawling load (bytes downloaded)



(b) CDF of DHT load (DHT traffi c)

Figure 6: Multiple fixed web sites: Crawling and DHT loads balanced over 80 nodes

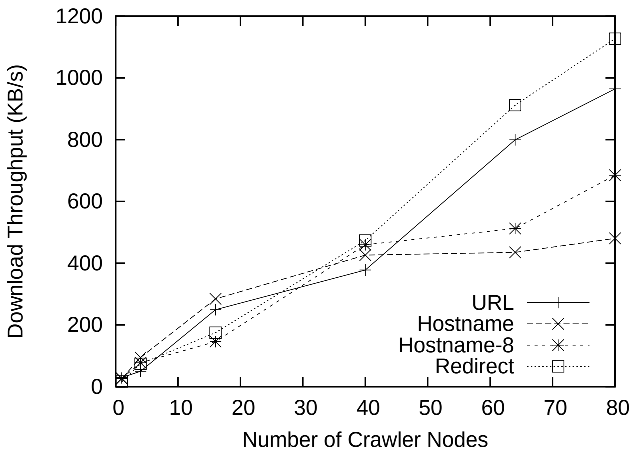


Figure 7: *Throughput (Multiple Fixed Sites)*

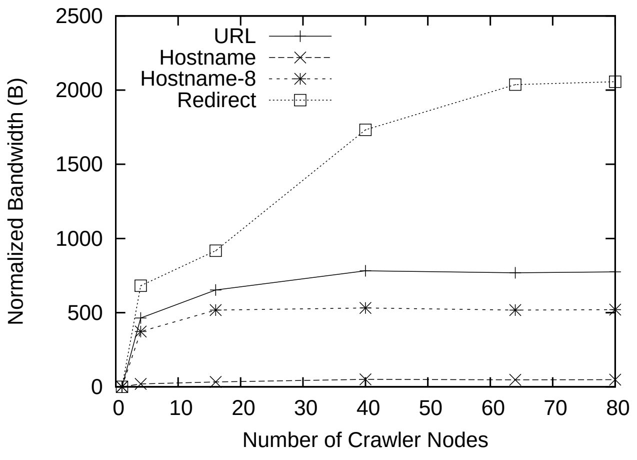


Figure 8: *Normalized DHT Load (Multiple Fixed Sites)*

one from the crawler node to the redirector node, and from the redirector node back to another crawler node. Coupled with the larger WebPage tuple used, the overheads are significant. In contrast, *Hostname* incurs low DHT load due to the high probability of self-links since we are constrained only to 45 web servers.

4.4 Crawl of Single Site

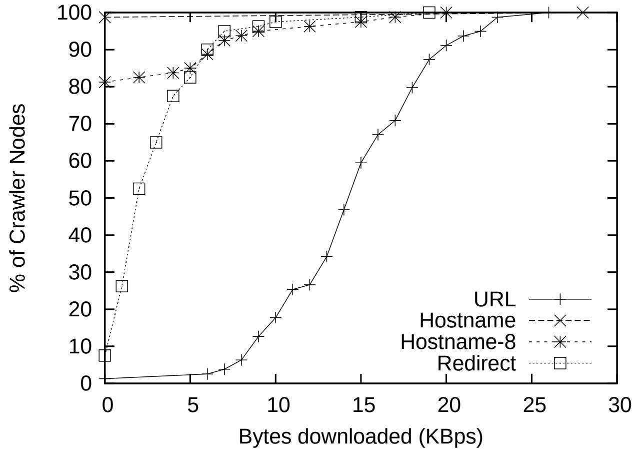
Finally, we study workload (C) which constrains a crawl to a single site, `http://groups.google.com`.

4.4.1 Crawling and DHT Load

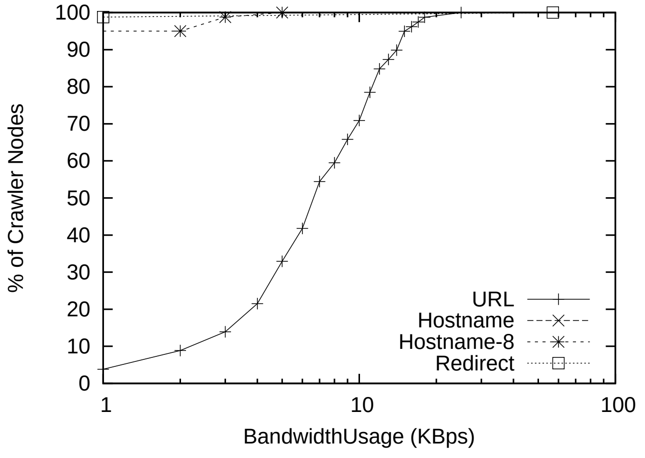
Since only a single site is being crawled, figure 9(a) shows extreme imbalance for *Hostname*, where only one crawler node is being utilized. Even with *Hostname-8*, 80% of the crawler nodes are idle. Figure 9(b) shows that only *URL* results a balanced DHT load across all the nodes.

4.4.2 Throughput

Figure 10 shows that *Hostname* has the lowest throughput, followed by *Hostname-8*. In both instances, the throughputs are poor due to the large number of idle crawler nodes. As the number of nodes increases, only *URL* scales linearly since there are no idle crawlers using this scheme. Surprisingly, even *Redirect* has poor throughput despite having a *URL* redirection mechanism that results in few idle crawlers. Since there is only one redirection node that is responsible for redirecting requests for `http://groups.google.com`, this quickly leads to bandwidth saturation of that node, which is made worse when the number of crawler nodes increases. In fact, figure 9 shows that the single redirector node incurs a DHT load of around 60KBps. Another reason for the saturation in



(a) CDF of crawling load (bytes downloaded)



(b) CDF of DHT load (DHT traffi c)

Figure 9: Single web site: Crawling and DHT loads balanced over 80 nodes

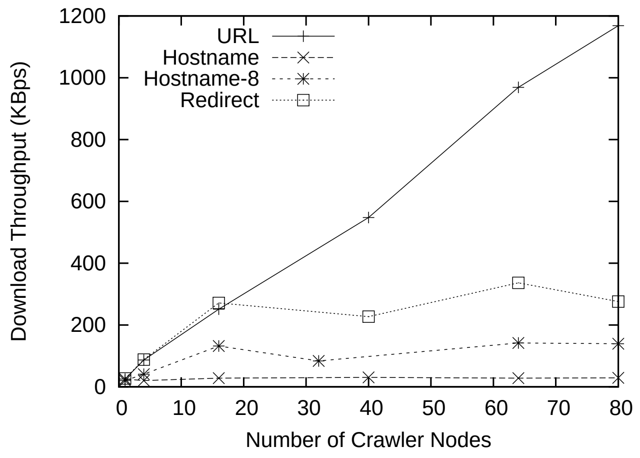


Figure 10: *Throughput (Single web site)*

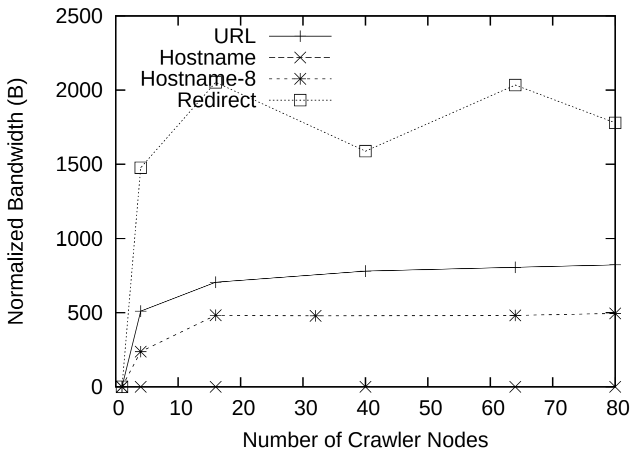


Figure 11: *Normalized DHT Load (Single web site)*

Redirect is due to the fact that new links from the redirection node are periodically flushed every second.

4.4.3 Communication Overheads

Figure 11 shows that *Hostname* incurs no DHT load since only one node is involved in the actual download, and only self-links are crawled. Similar to the earlier experiment, *Redirect* incurred up to 3 times as much bandwidth as *URL*.

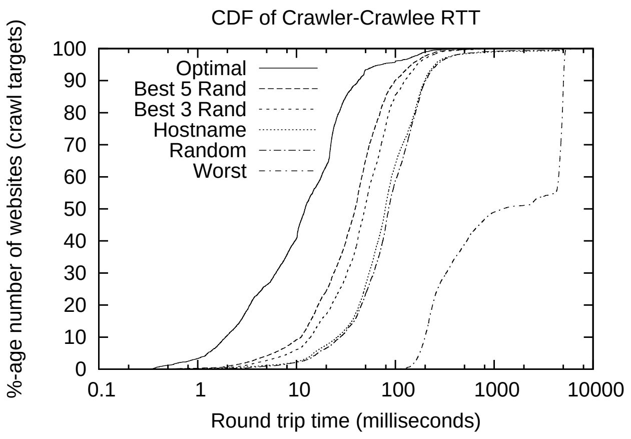
4.5 Network Locality

In this section we study the effects of network locality on our partitioning schemes. We sampled a set of hosts that were crawled in exhaustive crawl workload (A) and measured the round trip time (RTT) between each host and each PlanetLab node. This measurement was done offline using the `ping` utility. We averaged our results over two separate runs executed at different times on different days. In practice, `ping` only worked for 70 nodes out of all 80 nodes forcing us to restrict our sample only to those hosts assigned to these 70 nodes.

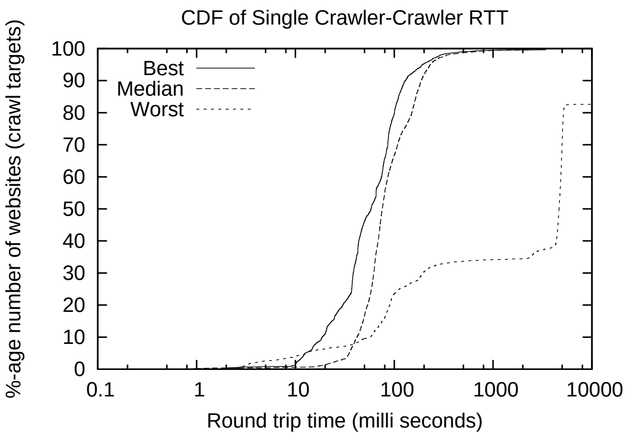
From this data, we considered a number of possible assignments of each host to crawling nodes:

1. *Optimal*: pick the best of all 70 crawlers
2. *Best 3(5) random*: pick the best of 3(5) random crawlers
3. *Random*: pick a crawler at random out of all 70 crawlers
4. *Worst*: pick the worst of all 70 crawlers
5. *Hostname*: pick the actual crawler chosen in executing workload (A)

In Figure 4.5(a) we show a CDF of the RTT for each host distributed over all 70 crawlers for each of the six schemes listed above. As expected the *optimal* assignment does much better than the *worst* assignment, and the actual *hostname* assignment shadows the *random* assignment very closely. For



(a) RTT of Crawlees from all Crawlers



(b) RTT of Crawlees from Single Crawler

Figure 12: Round-trip times between Crawlees and Crawlers

the median number of hosts the RTT of optimal and hostname is 12.19 msec and 79.1 msec respectively. In contrast, for the median number of hosts, the best of 5 and 3 random picks result in RTT of 39.18 msec and 49.13 msec respectively. This suggests that even a simple heuristic like best of 3 random picks can be a significant improvement over the random/hostname assignment.

As a sanity check we also considered the effect of running a centralized crawler on any one of the 70 crawlers. In terms of the average RTT for all hosts for each single crawler we determine the best, median and worst such crawler. In Figure 4.5(b) we show a CDF of the RTT for each host over the best, median and worst single crawler. The CDF of the worst single crawler is particularly instructive as it has a knee representing 60% of the hosts are very far away.

4.6 Summary

Scheme	Crawl Load	DHT Load	Crawlee Load	Network Locality	Comm. Overheads
URL	+	+	−	−	−
Hostname	−	−	+	?	+
Redirect	+	?	+	+	−−

Table 1: Summary of Partitioning Schemes

We summarize the different schemes in table 1. The row labels show the different schemes while the column labels show different desired features. An extra column is added for network proximity even though we did not explicitly compare the different schemes in our experiments. "+" is given when a feature is supported, and "-" otherwise. Values of "?" are assigned when it is unclear whether a feature is supported.

None of the schemes is a clear winner. *URL* achieves good load balancing at the expense of the other features. *Hostname* has low communication overheads, but as our experiments have shown, display poor load balancing in two out of three workloads, leading to poor throughput. It also does not provide support for network locality, and depending on luck, may end up on a crawler node that is close or far away from the crawlee.

Only *Redirect* provides support for both rate throttling and network proximity. However, it comes at a price of increased DHT communication overheads. As our experiments have shown, DHT load may also not be balanced if *Hostname* is used to determine the redirection nodes. However, if redirection is used together with *URL*, the DHT load can still be balanced.

5 Related Work

In this section we provide a quick survey of current high throughput web crawlers. Most of these crawlers are cluster-based solutions. These parallel crawlers deploy either a central coordinator [10] that dispatches URLs to be parallelized across a set of nodes, or uses hash-based schemes [12, 22]. In such environments, the focus is solely on high throughput crawling. These parallel crawlers are highly specialized and tuned to run essentially one crawl query. Since they are deployed "in-house", their bottleneck is usually the organization's incoming and outgoing bandwidth to the outside world. As their nodes are centralized, they cannot leverage geographic locality of crawlers to crawlees.

To the best of our knowledge, the only P2P crawler that has been deployed to date is Grub [3]. They use a SETI@home deployment model where there is a central URL dispatcher within Grub itself. Currently, they have a base of approximately 23993 users. As they require a central coordinator, they are not truly P2P, and the crawl process is entirely under the control of Grub.

Of most relevance to our work are DHT-based web crawlers built using Chord [24]. These crawlers propose a single partition scheme, and do not examine the tradeoffs between different schemes, nor do they investigate rate throttling of sites. Since they experiment with only a few local machines, it is unclear how well their system performs in practice when deployed over a distributed set of nodes. Lastly, their optimizations are derived from tight coupling with Chord, which would not be possible in a DHT agnostic design. Further, our query-centric approach ensures that our web crawler can run over any query processor, and is also easy for user customization.

User customizable centralized focused crawlers have been proposed previously. One notable example is the BINGO! [23] system. These systems tend to be too heavyweight to be deployed by the average home user. However, by harnessing the resources of nodes in the network, these systems would have the potential to scale and gain widespread adoption. In future, we will consider customized centralized focused crawlers into our system, by treating them as “black boxes” within our user customized operators.

6 Future Work

In this section we outline some of the interesting research problems we intend to explore in future:

- **Fault Tolerance.** While Bamboo is a churn-resilient DHT, we have not provided any mechanisms to ensure that the crawl is robust in the presence of node failures. There are a few possible solutions to increase fault tolerance. First, we can run redundant queries at a high cost. Second, we can store the intermediate state of query execution in a highly available infrastructure like OceanStore that transparently handles replication. Last, we can implement and utilize fault tolerant operators such as FluX [21].

- **Single Node Throughput.** In this implementation, we have not made any efforts to improve the throughput of a crawl on a single node. This has an impact on the absolute throughput achievable by the crawler. For example, our TeSS implementation is CPU intensive and does not perform well under heavy load on PlanetLab. Most high-performance parallel crawlers [22, 12] provide mechanisms to increase crawler throughput even on a single node. Such techniques are orthogonal to the crawl distribution techniques that we have studied, and we will explore applying some of their optimizations into our system.

- **Mid-query Relevance Feedback.** There has been work done by the database community on providing mid-query relevance feedback [13]. This is essential for long-running focused crawlers where users may wish to alter the crawl ordering on-the-fly based on intermediate results. Ideally, we would like to provide this support generically within PIER itself using techniques from the CONTROL [13] project.

- **Continuous Aggregation Query.** The *partition by URL* scheme suffers from the lack of a central control point for per-server rate throttling. It is interesting to see if we can execute a continuous in-network aggregation query using PIER to approximate the aggregate per-server download rates at runtime. While the cost of such a continuous query may be expensive, we do not require exact answers and even a coarse estimate may be sufficient and cheaper than using redirection.

- **Load Balancing.** Thus far, we have only considered load balancing mechanisms using either the DHT or via redirection at a higher level. There have been several proposals for load-balancing DHTs [19] with the use of virtual servers and other techniques. Since PIER is DHT agnostic, in principle, we should be able to experiment with our crawler on top of DHTs with such advanced load-balancing capabilities.

- **Declarative Load-Balancing and Network Proximity.** An interesting question is if we can get load-balancing and network proximity to work ”auto-magically” as part of a PIER optimization process. For example, rate limiting can be expressed as a query rewrite, by turning the ”DHT communication overheads” into an explicit part of the query. If this is possible, our crawler would be totally declarative, including declaring the performance constraints[3].

- **Complex Queries.** We will explore supporting more complex crawl queries besides the link extraction query described in Section 3. Examples of such queries include distributed page rank computation, computing web site summaries and constructing inverted indexes over the web pages.

- **Web Crawler Service.** Our eventual goal is to deploy the distributed crawler as a long running service on PlanetLab and integrating it with existing focused crawlers [23]. In actual deployment, we will have to tackle the issues of *sharing work* between crawl queries issued by different users. It is conceivable that this sharing can be provided by generic sharing techniques used in continuous queries systems [11].

7 Conclusion

This paper presents an initial exploration of the design space of building a distributed web crawler over DHTs. We have provided the motivations of a distributed web crawler that can serve as a basic building block for more advanced decentralized search applications. Our crawler is designed to run over any DHT, and by expressing crawl as a query, we permit user-customizable refinements of the crawl easily, as well as enabling the use of other query processors such Telegraph, as long as they support recursion.

In designing the crawler, we have identified important tradeoffs in different crawl execution strategies, and validated

[3]This idea has been put across to us by Joseph Hellerstein.

these tradeoffs via experiments using PlanetLab and querying live web sources using different workloads. We have also quantified the benefits of network proximity improvements that can be gained by exploiting the location of the distributed set of nodes on PlanetLab.

8 Acknowledgements

We would like to thank Joseph Hellerstein, Anthony Joseph, John Kubiatowicz, Mehul Shah, Ion Stoica, and other members of the PIER and Berkeley database group for giving us valuable feedback on this work. We would also like to thank Ryan Huebsch for PIER software support, and Karthik Lakshminarayanan for providing us with the ping measurement software that we used to obtain our RTT measurements on PlanetLab.

References

[1] Gnutella. http://gnutella.wego.com.

[2] Google. http://www.google.com.

[3] Grub's Distributed Web Crawling Project. http://www.grub.org/.

[4] Kazaa. http://www.kazza.com.

[5] PlanetLab. http://www.planet-lab.org/.

[6] SETI@home. http://setiathome.ssl.berkeley.edu/.

[7] Study Tallies Sites Blocked by Google. http://query.nytimes.com/gst/abstract.html?res=F3081EFD3B580C768EDDA90994DA404482.

[8] TeSS: The Telegraph Screen Scraper. http://telegraph.cs.berkeley.edu/tess.

[9] The Deep Web: Surfacing Hidden Value. http://www.press.umich.edu/jep/07-01/bergman.html.

[10] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[11] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Datafbw Processing for an Uncertain World.

[12] J. Cho and H. Garcia-Molina. Parallel crawlers. In *Proc. of the 11th International World–Wide Web Conference*, 2002.

[13] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *Proceedings of ACM SIGMOD*, pages 171–182, 1997.

[14] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *Proceedings of VLDB Conference*, September 2003.

[15] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*. ACM, November 2000.

[16] J. Li, B. T. Loo, J. Hellerstein, F. Kaashoek, D. Karger, and R. Morrris. On the Feasibility of Peer-to-Peer Web Indexing and Search. In *IPTPS 2003*.

[17] B. T. Loo, R. Huebsch, J. M. Hellerstein, T. Roscoe, and I. Stoica. Analyzing P2P Overlays using Recursive Queries. *UC Berkeley Tech Report, UCB//CSD-04-1301*, Jan 2004.

[18] R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1993.

[19] A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured p2p systems, 2003.

[20] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. *UC Berkeley Technical Report UCB//CSD-03-1299*, Dec 2003.

[21] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An adaptive partitioning operator for continuous query systems. 2003.

[22] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *ICDE*, 2002.

[23] S. Sizov, M. Biwer, J. Graupmann, S. Siersdorfer, M. Theobald, G. Weikum, and P. Zimmer. The bingo! system for information portal generation and expert web search.

[24] 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 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[25] C. Tang, Z. Xu, and M. Mahalingam. pSearch: Information retrieval in structured overlays. In *ACM HotNets-I*, October 2002.