CIS 555 Final Project

Poogle: Penn's Favorite Search Engine

Sanjay Paul, Levi Cai, Dan Kim, and Federico Nusymowicz
{sanjayp, cail, dki, fnusy}@seas.upenn.edu

Professor Andreas Haeberlen
ahae@seas.upenn.edu

Abstract

Poogle is a search engine that runs on a distributed network of AWS machines. This paper describes, analyzes and evaluates the Poogle system.

I. Introduction

A. Project Goals

We built Poogle with certain objectives in mind. Our primary goals included:

- Efficiently crawling a large corpus
- Responding to several concurrent queries in a timely fashion
- Providing a clean user interface
- Serving up high-quality search results
- Developing a highly scalable system able to operate across a large network of peers

B. High-Level Approach

Our system operates in three distinct phases. Phase 1 consists of a process that crawls the web, and then indexes pages as they become available. The second phase calculates PageRank for all indexed content and readies the system to answer queries efficiently. Finally, a third process instantiates a server that answers user queries by calculating document rankings and returning search results in order of relevance. Figure 1 depicts the high-level approach.

Each of the processes relies upon a distributed network of peer nodes. Individual nodes are responsible for both running the modules and for storing portions of the database. We explain our architecture in further detail in Section II.

C. Project Timeline

- 4/18/2012: crawler operational.
- 4/21/2012: indexer operational.
- 4/28/2012: crawler and indexer integrated; combined module operational across variable-sized node networks.
- 4/29/2012: user interface completed.

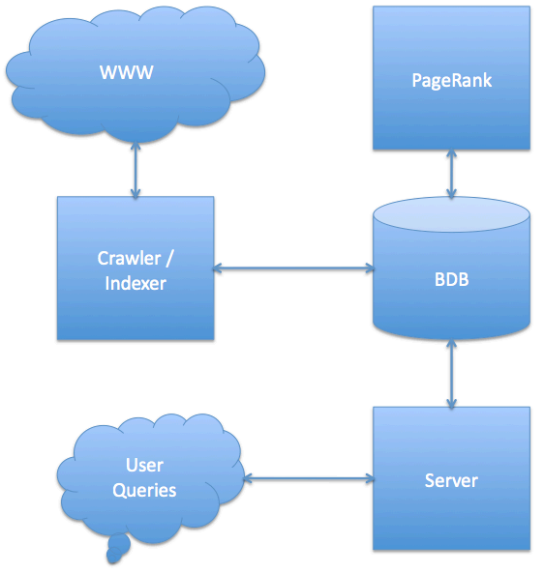

Figure 1: High-Level Approach

- 5/3/2012: PageRank operational.
- 5/5/2012: server module functional.
- 5/6/2012: tuned ranking function.
- 5/8/2012: finished integrating components.

D. Division of Labor

We worked together as much as we could and pair-programmed relatively often, which helped ease component integration towards the end of the project. Each team member contributed to most modules. We also assigned individual accountability for certain specific tasks-

Sanjay Paul: crawler functionality and Pastry ring management.

Levi Cai: indexer functionality and Pastry network testing.

Dan Kim: user interface and PageRank module.

Federico Nusymowicz: server module and BerkeleyDB/AMI management.

II. Architecture

A. Database

We store our data persistently using BerkleyDB. Our implementation distributes BDB data structures across several FreePastry peer nodes, where each peer node takes responsibility for a subset of the data. Our most relevant data structures include:

- *Pages*, which contain the raw XML/HTML content downloaded from a given URL, PageRank information, and a list of the page's outgoing links. Each peer node takes responsibility for a subset of URL hosts.
- *HitLists*, which mimic the Google data structure going by the same name [1]. Each HitList corresponds to a specific word within a Page. HitLists also count the number of times the term occurs within a document, maintain position information for each word occurrence, and hold additional term-ranking information, such as whether the term was a 'fancy hit' (i.e. a title, header, or meta information).
- *HitBins*, which aggregate all the HitLists for a specific word. Individual nodes in the network take responsibility for storing a subset of word HitBins.

As shown in Figure 1, BDB data structures serve as the main point of interface between our three component processes.

B. Crawler / Indexer

Our crawler's architecture borrows heavily from Mercator's design [2], with some simplifications made in the interest of reducing development overhead. Each crawler node operates as follows:

1. Poll the local BDB's URL queue.
2. Enforce politeness; if the URL's host was recently pinged, move the URL to the back of queue and repeat step 1.
3. Download the HTML/XML content and store it as a Page in the local BDB.
4. Extract all links and route each one to the node responsible for the URL's host.
5. Go back to step 1.

Whenever a crawler node receives a link, it checks whether the URL is a duplicate, and if so

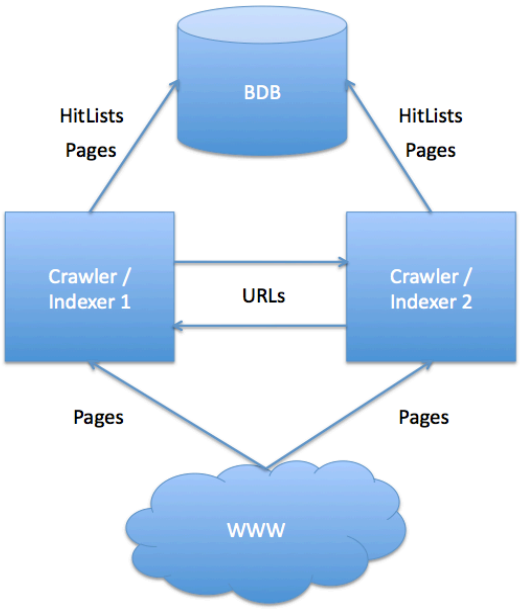

Figure 2: 2-Node Crawler/Indexer

the link gets discarded. Otherwise the link gets added to the back of the node's URL queue.

As soon as pages become available, the indexer parses them one by one. Parsing entails forming HitLists for each word in the content body and then calculating the term's TF factor. Figure 2 depicts a 2-node crawler & indexer process.

We designed our crawler / indexer with fault tolerance in mind. The process regularly checkpoints by syncing with disk, thereby allowing the crawler and indexer to pick up from where they stopped in the event of a crash. The process is also highly scalable and stable – we successfully ran it across 10 nodes and crawled continuously with no problems.

Our Phase 1 architecture's main drawback regarding scalability was the fact that new nodes could not dynamically join or leave the process without impacting the overall network's execution. A possible future improvement could involve periodic checks to dynamically redistribute Pages.

After giving the process its shutdown signal, the crawler stops running and the indexer sorts HitBins by TF factor, in order to later improve our servers' query response speeds.

B. PageRank

After halting the crawler / indexer process, our system moves on to compute PageRank. The

PageRank calculation begins by aggregating Page data at a single master node. This allowed for better ordering of our results, enabling us to return more authoritative, reliable sources. We implemented PageRank using an iterative, pseudo-distributed Hadoop job. The results from crawling were then aggregated and fed into Hadoop.

Since calculating a URL's PageRank relies on the rank of pages that link into that URL, and since outgoing links compose other page's ranks in turn, we needed to iterate a large number of times in order to come to an acceptable result.

Our map algorithm accepted URLs along with their associated page ranks and outgoing links. Then, in the combining stage, we aggregated an entry's PageRank by adding the scaled ranks of all incoming URLs. We iteratively used these entries in the reducing stage to calculate page ranks until reaching some degree of convergence.

Defining the convergence function proved to be a fairly non-trivial task. The iterative reduce step took a long time to run (hours), and as an additional challenge, determining the proper threshold for 'convergence' proved to be more of an art than a science. In the interest of time, we chose a less sophisticated approach: set number iteration. We found that although set number iteration proved more inaccurate than a rigid definition of 'diff' convergence, with a sufficient number of iterations, the end values varied little enough to consider them fairly accurate PageRanks. As an added benefit, set number iteration helped us predict the process' runtime.

Once the PageRank algorithm completed, we distributed PageRank scores across all network nodes and appended them to HitLists in order to later improve server response time.

C. Server

After all HitLists got updated with their relevant PageRank scores, we switched each node to server mode. Servers then began actively answering queries. More specifically, servers:

1. Listened for query requests.
2. Split queries by term and requested the corresponding HitBins from other servers.
3. Waited for HitBins to return and cached HitBins for the most popular terms. Servers only retrieved the top 10,000 entries (based on each entry's TF factor) from a given HitBin in order to improve retrieval speed.
4. Servers then calculated document ranking based on an augmented TF-IDF vector model.
5. Finally, servers returned the query results.

We implemented a Tomcat servlet in order to route queries to our servers. The queries themselves then got hashed and routed through the Pastry ring, thus balancing computing load across all server nodes and improving mean response time.

Routing queries through Pastry provided the unexpected side effect of fault tolerance, since even in the event that a server node crashed, the search engine still remained operational.

D. Ranking

Every returned HitBin contains HitLists sorted by TF factor, which the indexer computes as:

$$\text{freq(word,doc)} / \max[\text{freq(any word,doc)}]$$

…where word counts are double-weighted for each of the 'fancy hits' described earlier. Servers also gather the HitBin's total size (n) when they retrieve the bin's top 10,000 entries. Additionally, servers communicate with each other at startup in order to calculate the total size of the corpus (N). To determine a HitList's TF-IDF weight, servers calculate:

$$\text{TF-IDF} = \text{TF} * \log(N/n)$$

Servers then weigh each query term according to the formula:

$$w_{query}(\text{word}) = 0.5 + [0.5*\text{freq(word,query)} / \max(\text{freq(any word,query)})] * \log(N/n)$$

For each Page referred to in one of the retrieved HitLists, servers then calculate the cosine similarity between the query and the page, and scale by the Page's PageRank in order to calculate the final ranking score.

III. Evaluation

A. Crawler / Indexer

The crawler and indexer were run as a coupled pair in a single JVM process on each of ten total Amazon Machine Instances on the EC2 cloud. They were allocated a virtual memory size bound in the range between 1 and 2 MBs (min/max). Overflow data was pushed to the key-value store provided by the open-sourced Berkeley DB distribution and top-level caching (as well as that provided by Berkeley DB) was exploited to improve performance. Resource allocation was distributed across the myriad and performance-throttling resource sinks present in each process, and these were primarily directed towards tracking duplicate encountered URLs (cached), the URL expansion frontier (breadth-first), and space occupied by indexer entries en route between

machine nodes and from main memory to disk storage. Each component was allocated an explicitly bound capacity that varied by priority and relative speed.

The internal construction of the crawler-indexer reflected their outward performance characteristic in that the crawler thread pool was relatively small (nine total was found to be an optimal value) and the indexer thread pool much larger (thirty threads was used though less testing was done to pinpoint an optimal). This follows naturally from the fact that much of the crawler's operation is network-IO bound due to intra-node URL passing and page downloading with some unavoidable disk overhead due to a non-negligible cache miss rate (i.e. often >20%, but still tolerable due to locality). On analysis, the average crawl spent less than 10% of its time performing computations – the rest was due to blocked I/O. By contrast, the indexer had much more opportunity to exploit parallelization due to its inherently compute-bound primary operations. An addendum in this regard was the fact that indexed buckets needed to be shuffled between machine nodes, resulting in large messages being directly routed to nodes (based on hash).

Aside from using a SLRU cache replacement policy on important top-level objects, we managed to achieve high performance by adopting a "route-to-final" policy – in effect, any object (i.e. indexer entry, URL, etc) that was to be pushed from a source to a sink within the system was buffered and hastily evicted to its final destination. The objective here was to clear space in memory by making some bandwidth concessions and to minimize time wastage caused by objects pending a push to the endpoint machine instance. This was an exceedingly effective practice, though it led, not surprisingly, to the side effect of high-frequency message passing and overflow within the system. The FreePastry distributed hashing scheme utilized to coordinate the machines proved to be less tolerant to high network traffic than we anticipated and we had to implement coherent buffering, throttling, and node-rotation schemes in order to achieve robustness, which was strained by adverse message queue overflows and locking exceptions due to timeouts. While buffering (with direct message-passing) and throttling were fairly intuitive countermeasures, we also chose to cycle messaging allowance by triggering nodes to flush their buffers to the system in sequence.

The crawler and indexer performed well above expectation with an average throughput of 160 pages crawled and 117 pages indexed per minute. In a crawling run of approximately 100 minutes, we achieved an indexed corpus size of about ~128K pages. The story, however, was markedly different on an individual machine-to-machine basis. In short, many machines went massively under-utilized due to discrepancies caused by inconsistent hashing. Whereas the machine nodes attempt enforce a uniform hash, the distribution of hashed content (in this case of the URLs and keywords) achieved a perceptible skew. Future utilization of the crawler-indexer might consider improving its behavior by either virtualizing nodes (and allowing dynamic reassignment of hashes) and/or leveraging machine learning principles to improve the system's anticipated distribution of hashed content.

B. Server

Our system was designed to provide fast results, where much of the system's time was spent on pre-processing results. As a result, each page on average had about 80B of associated data, after the nearly 1.5 hour crawl and indexing session, the fastest server had nearly 2.2GB of data stored. Result retrievals of single words were near instantaneous regardless if it was stored in the cache or not, and multi-word searches were not much longer.

"This is the coolest place on earth" returned in approximately 2.7 seconds and when queried again it returned on average in 1.2 seconds due to the cache. Several other 7+ word queries returned at similar time intervals with over several hundred results each. Single word entries are much faster.

IV. LESSONS LEARNED

Building a search engine was an incredibly rich experience – there were countless opportunities for optimizations, tweaks, and improvements. Some of our ideas proved so interesting that we often had trouble focusing on completing the project's most basic features. We focused so much on optimizing our crawler, for example, that by the end of the project we barely had time to refine our PageRank algorithm. If we had the chance to start over, we would definitely get all the basic features working on a basic level before starting to polish any part of our code base.

We also learned about the importance of clear interfaces. After our first few nights of programming together, we all thought we had a relatively thorough understanding of each component's architecture and required data structures. After splitting the work, however, we quickly realized how wrong we were. Lack of

clearly specified interfaces ended up costing us countless hours of integration effort.

IV. Conclusion

In summary, we unanimously concur that architecting this system was by far the most challenging task any of us have ever undertaken, but also among the most rewarding. We gained an appreciation for a wide breadth of challenges inherent to massively scaled system design and the solutions they necessitated. On a fundamental level, a search engine handily incorporates almost every relevant facet of distributed system design, ranging from parallelization considerations to robustness to scalability and high-performance operation. The lessons we take away from the experience will undoubtedly carry with us well into the future.

References

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

[2] M. Najork and A. Heydon. "High-Performance Web Crawling". Kulwer Academic Publishers, Inc. Compaq SRC, September 2001.