CLIP: A Compact, Load-balancing Index Placement Function

Michael McThrow

Storage Systems Research Center
University of California, Santa Cruz

Abstract

Existing file searching tools do not have the performance or accuracy that search engines have. This is especially a problem in large-scale distributed file systems, where better-performing file searching tools are much needed for enterprise-level systems. Search engines use inverted indices to store terms and other metadata. Although some desktop file searching tools use indices to store file metadata, they are not designed for large-scale distributed file systems. We propose bringing inverted indices to the distributed object-based file system named Ceph via CLIP: a compact load-balancing index placement function that distributes the inverted index across the file system. The inverted index is distributed based on the frequency of terms of the files in the file system.

1. Introduction

File searching, especially in large-scale file systems, is slow and gives inaccurate results. Compared to searching for web pages on the World Wide Web, file searching offers neither the accuracy nor the performance that search engines have. There are some key differences between searching for web pages and searching for files, which explains why web searching provides greater performance and more accurate results than file searching. First, web pages typically link to other web pages, whereas file systems typically do not support such a relationship between files, although there has been research in bringing such support to file systems [1, 7, 8, 9]. Secondly, the number of documents for a relevant query in a web search is much larger than the number of files in a file search, which usually corresponds to just one file. Because of the much smaller number of relevant documents, file searching needs to be more exact than web searching.

One reason why search engines perform well is the use of inverted indices. Rather than traversing the Web to search for a query, search engines search the inverted index. An inverted index is a data structure that consists of terms, the frequency of those terms, references to the web pages that specific terms are stored in, and the position of specific terms in those web pages [12]. Documents are weighted based on the amount of terms per document, and terms are weighted based on the amount of times they appear per document and the amount of documents that they appear. Given a query q that consists of t terms, the top r matching documents are ranked according to the weights [12]. Zobel and Moffat provide a wealth of further information about inverted indices [12].

We shall turn away from file searching for a moment and discuss the Ceph file system. Ceph is a petabyte-scale distributed file system developed by the Storage Systems Research Center that consists of object storage devices (OSDs). An OSD consists of a processor, memory, and a hard disk. Ceph pseudo-randomly places data across OSDs using the CRUSH algorithm [10]. CRUSH is a “scalable pseudo-random data distribution function designed for distributed object-based storage systems that efficiently maps data objects to storage devices without relying on a central directory” [11]. CRUSH is compact, which turns out to be essential for the scalability of Ceph [11].

Since we want to improve the performance of file searching in Ceph and to improve the quality of our searching results, we propose using an inverted index for searching. We want the inverted index (which will be known as simply the *index* through the rest of the paper) to be distributed evenly across the OSDs such that specific OSDs are not overloaded with queries containing very popular terms. We know that search queries are skewed in a Zipf-like distribution. Because certain terms are more popular than others, we want to distribute the index such that the load of terms is balanced across the OSDs based on the term's frequency.

We introduce CLIP, a compact load-balancing index placement function. CLIP is an extension of CRUSH that handles skewed update profiles while retaining its compactness. CLIP does not require storing large data structures containing the entire distribution of terms. Instead, a power function is produced from the distribution of terms, and a small data structure maps these terms to their corresponding k value given a rank r, based on the power function. Based on the frequency of a term, CLIP computes how many OSDs are needed to store that term, in order to balance the load.

The rest of the paper is structured as follows. Section 2 describes work related to CLIP. Section 3 further describes the problem of distributing the index across the OSDs. Section 4 describes how CLIP works. Section 5 shows the results of our work, and Section 6 provides a conclusion of our research results, as well as our future work.

2. Related Work

There are file searching tools designed for personal computers, such as Google Desktop and Apple Spotlight, that maintain an index of hard disks and return a ranked list of files that match a query [2, 5]. These tools perform much greater than file searching tools that traverse the directory tree. However, the key issue to applying such tools to Ceph is that these tools are not scalable and are not designed for distributed, petabyte-scale file systems.

There is an effort to make file searching more contextual rather than based solely on content. Soules and Ganger proposed adding extensions to file systems containing metadata from applications and inter-file relationships [8]. They built a file searching tool called Connections, that uses temporal locality to infer relationships between files [9]. Shah et al. extended this work by adding information about the provenance of a file, and by using causality instead of temporal locality to contextually search for files [7].

3. Problem

In order to understand how the update stream is skewed, we need to describe Zipf's law and Zipf distributions. Zipf's law is an observation named after linguist George Kingsley Zipf that determines the frequency of specific words in an English text [6]. Given the rank r of a word, Zipf's law states that the frequency of the word is inversely proportional to r:

$$f_i \propto \frac{1}{i^{\alpha}} \, (r = 1, \ldots, N)$$

where f is the frequency of the term, $\alpha \approx 1$, and N is the total number of words in the corpus [6]. According to this law, the most common word occurs twice as often as the second common word, the second most common word occurs twice as often as the third most common word, and continues until the least common word is reached.

A Zipf distribution is defined as "a set of values that follows Zipf's law" [6]. The distribution function for a Zipf distribution is

$$f_i = \frac{K}{i^{\alpha}}$$

where K is the frequency of the most popular term, and α is the relative popularity of terms in the distribution [6]. When graphing a Zipf distribution on a log-log scale, the graph appears linear. This means that there are few terms that occur often, and there is a large amount of terms that occur rarely.

Knowing that the update stream is skewed in a Zipf-like distribution, the problem is how to load-balance the update stream across the OSDs such that each OSD is not overloaded with popular terms. Since it is essential for Ceph's scalability that any data allocation function is compact, we need to make sure than an index allocation function is compact as well. We could think of two approaches: using a B-tree over fixed-sized objects, and by using a hash function to assign terms to OSDs. B-trees use efficient splitting and joining operations to keep the paths to each leaf at approximately equal length. We think that B-tree mechanisms can be adapted to provide additional balancing in terms of index updates. The hash function approach uses CRUSH to assign terms to OSDs. We ultimately decided to pursue load-balancing via a hash function, which is further described in Section 4.

4. Approach

4.1. Dividing the Index

Since certain terms are more popular than others due to Zipf's law, the index needs to be divided across the OSDs such that individual OSDs are not overloaded with queries containing very popular terms. Given a term and a value k, which is based on the relative term (update) frequency and the total number of OSDs in the file system, CRUSH maps $(term, k)$ to a set of OSDs $\{OSD_1, \ldots, OSD_k\}$. Popular terms will have a relatively large value of k, while $k = 1$ for rare values, which conforms to Zipf's law. The term is then randomly assigned to an OSD within that set. To find a specific term in the file system, the set of OSDs that map to that term are queried. This assignment of terms to OSDs is illustrated in Figure 1.

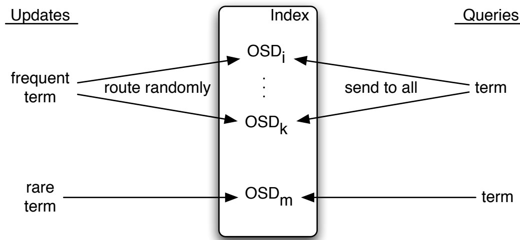

Figure 1. Routing of updates and queries to OSDs

4.2. Keeping Track of Terms

One problem is storing the distribution of terms so that essential values such as k can be calculated. Mapping each term to its k value is expensive and would destroy the compactness of the index placement function. We want to categorize terms based on k and only store the Bloom filters instead of all terms. Our approach was to use Bloom filters to categorize terms based on k.

4.2.1. Categorization with Bloom Filters The National Institute of Standards and Technology's Dictionary of Algorithms and Data Structures defines a Bloom filter as "a probabilistic algorithm to quickly test membership in a large set using multiple hash functions into a single array of bits" [3]. Because Bloom filters are probabilistic, this means that there is a risk of obtaining false positives in membership tests. The probability of a false positive occurring is

$$\left(1-\left(1-\frac{1}{m}\right)^{\beta n}\right)^{\beta} \approx \left(1-e^{\beta n/m}\right)^{\beta}$$

where m is the size of the bit array (in bits), n is the number of members in the Bloom filter, and β is the amount of hash functions [4]. Increasing the size of the bit array m, as well as increasing the amount of hash functions β, decreases the probability of a false positive occurring, but also increases the amount of computation required [4].

For large sets of data, Bloom filters are more compact than other data structures such as B-trees. To mitigate the effects of false positives, we store the top 0.1% of the distribution in a B-tree. The rest of the distribution from after the first 0.1% to the last entry where $k > 1$ is split logarithmically into s sections, each represented by a Bloom filter. Each Bloom filter consists of terms that correspond to a specific range of k values, and each term in that filter has an *estimated length* of the length of the highest ranked term in the range.

5. Results and Evaluation

5.1. Bloom Filter Results

Our experiments for our tests involving Bloom filters was implemented and evaluated on a 1.83GHz Intel Core Duo computer with 512MB RAM running Mac OS X 10.4.9. The Python programming language was used in our experiments, and the version of the Python interpreter used is Python 2.3.5. We used the PyBloom library, a public-domain Bloom filter library implemented in Python.

Bloom Filters	$s = 3$		$s = 15$	
Bloom Filter Sizes	$\{9, 89, 792\}$		$\{1,1,1,1,1,1,1,1,1,1,1,1,9,89,780\}$	
Variables	$m/n = 6, \beta = 4$	$m/n = 10, \beta = 7$	$m/n = 6, \beta = 4$	$m/n = 10, \beta = 7$
False Positives	$\{0,3,0\}$	$\{0,0,0\}$	$\{0,0,14,0,0,1,0,0,0,0,3,22,0,3,0\}$	$\{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0\}$
Execution Time	20.608s	24.823s	45.020s	48.245s

Table 1. Results of Bloom filter experiments using *Pride and Prejudice* as the corpus

We ran four experiments based on the m/n ratio (see Section 4.2.1 for definitions of m and n), the number of hash functions β, and the number of Bloom filters s. The corpus used in these experiments is from the book *Pride and Prejudice*, which contains 13,699 unique words. Table 1 shows the results of these experiments. Increasing the size of the m/n ratio, as well as increasing k, eliminates false positives. Increasing the number of Bloom filters have a greater impact on the time it takes to complete the experiment than increasing the m/n ratio and increasing k. However, we notice that some Bloom filters had sizes of one and nine. This is an issue, since it is a waste of space and time to maintain a Bloom filter for a miniscule amount of terms.

Performance also became a big issue when running the much larger corpus containing all of the documents from the Gutenberg Project 2006 DVD, which contains 690,369 unique terms. Some of these experiments took over ten minutes to run using this corpus. The number of terms that mapped to $k > 1$ was still small. To figure out how the number of terms scale with the size of the file system, we fitted the term frequency distribution (see Figure 2) to a power function and solved it for the rank r of a term, given $k = 1$ (see Figure 3). This confirms that keeping track of terms with $k > 1$ explicitly, i.e. extending CRUSH with a sorted data structure optimized for looking up terms, will still keep CRUSH compact.

5.2. Power Function Evaluation

We created a power function that fits to the distribution of words in the Gutenberg Project 2006 DVD corpus. The power function is $f_r \approx 0.2327r^{-1.1292}$, where f_r is the relative term frequency. Figure 2 displays the fitted power function and the actual distribution. In order to find out how many terms map to $k > 1$, we need to find out the highest rank r for which $k = 1$. For this we substitute $1/totalOSDs$ for f_r and solve for r, calculating how many terms in the distribution require multiple OSDs to be stored (i.e., when $k > 1$). The resulting equation is $r \approx \left(\frac{1}{0.2327 * totalOSDs}\right)^{-1/1.1292}$. Figure 3 shows a graph of this equation. If we take a linear approximation of this

equation, it shows that only 7-8% of the total number of terms require more than one OSD to be stored. This result is important, because only a relatively small number of terms needs to be stored in order to create the power function.

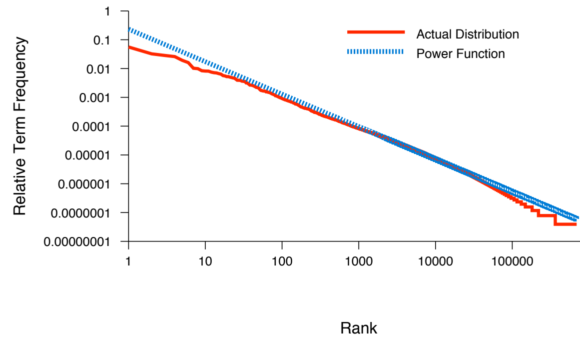

Figure 2. Term frequency distribution and fitted power function

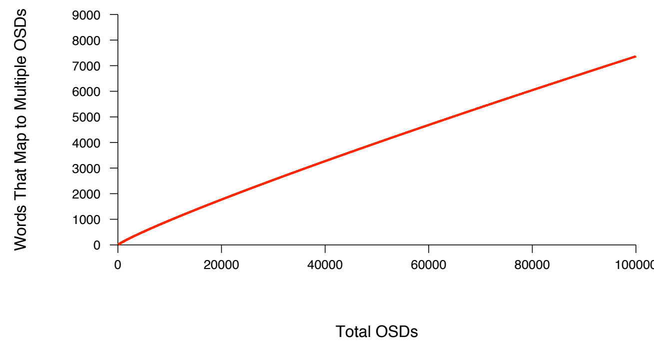

Figure 3. Estimating number of terms that map to multiple OSDs

6. Conclusions and Future Work

Even in very large systems that consists of hundreds of thousands of OSDs, by using a power function to approximate the distribution, only a relatively small number of terms require more than one OSD to be stored. This result makes querying terms easy, since rare terms are mapped to just one OSD, and common terms are mapped to a set of OSDs; queries do not have to search the entire file system. Storing the terms where $k > 1$ and their relative frequencies still lead to a compact placement function.

Our approach of using Bloom filters to categorize terms by their frequency did not work for three reasons. For one, since we developed a way to store a small data structure that maps terms to a corresponding k value, storing Bloom filters is unnecessary and expensive both in time and space. Secondly, the amount of terms is too small for Bloom filters to be useful. Finally, false positives can lead to significant communication overhead, since a term would allocate more OSDs than it actually needs, making queries much slower.

We need to verify that CLIP balances the load. We also must figure out how to manage the insertion and removal of terms in CLIP. File systems change over time, and CLIP must handle changes in the file system without major

performance issues. After these problems are solved, CLIP will be integrated into the Ceph file system. We need to investigate specific data structures to store the index across the OSDs, and also investigate how the distribution will be stored.

7. Acknowledgments

Carlos Maltzahn, Neoklis Polyzotis, and Scott Brandt were the main contributors to this project. Sage Weil provided the idea of using a B-tree over fixed-sized objects to load-balance the update stream.

This work was completed as part of UCSC's SURF-IT summer undergraduate research program, an NSF CISE REU Site. This material is based upon work supported by the National Science Foundation under Grant No. CCF-0552688 and CCF-0621534.

References

[1] Sasha Ames, Nikhil Bobb, Kevin Greenan, Owen Hofmann, Mark W. Storer, Carlos Maltzahn, Ethan L. Miller, and Scott A. Brandt. LiFS: An attribute-rich file system for storage class memories. In *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, May 2006.

[2] Apple, Inc. Working with Spotlight. http://developer.apple.com/macosx/spotlight.html, June 2006.

[3] Paul E. Black. "Bloom filter". in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology, 18 June 2007. (accessed 30 August 2007) Available from: http://www.nist.gov/dads/HTML/bloomFilter.html).

[4] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[5] Google. Google Desktop - Features. http://desktop.google.com/features.html, 2007.

[6] Vivek Sawant. Zipf's Law, Zipf Distribution: An Introduction. http://www.cs.unc.edu/ vivek/home/stenopedia/zipf/, 2004.

[7] Sam Shah, Craig A. N. Soules, Gregory R. Ganger, and Brian D. Noble. Using Provenance to Aid in Personal File Search. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 171–184, Santa Clara, CA, USA, June 2007.

[8] Craig A. N. Soules and Gregory R. Ganger. Why can't I find my files? New methods for automating attribute assignment. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 20–20, Berkeley, CA, USA, 2003. USENIX Association.

[9] Craig A. N. Soules and Gregory R. Ganger. Connections: using context to enhance file search. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 119–132, New York, NY, USA, 2005. ACM Press.

[10] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *USENIX'06: Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 22–22, Berkeley, CA, USA, 2006. USENIX Association.

[11] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: controlled, scalable, decentralized placement of replicated data. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 122, New York, NY, USA, 2006. ACM Press.

[12] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.