Indexing The World Wide Web: The Journey So Far

Abhishek Das
Google Inc., USA
Ankit Jain
Google Inc., USA

ABSTRACT

In this chapter, we describe the key indexing components of today's web search engines. As the World Wide Web has grown, the systems and methods for indexing have changed significantly. We present the data structures used, the features extracted, the infrastructure needed, and the options available for designing a brand new search engine. We highlight techniques that improve relevance of results, discuss trade-offs to best utilize machine resources, and cover distributed processing concept in this context. In particular, we delve into the topics of indexing phrases instead of terms, storage in memory vs. on disk, and data partitioning. We will finish with some thoughts on information organization for the newly emerging data-forms.

INTRODUCTION

The World Wide Web is considered to be the greatest breakthrough in telecommunications after the telephone. Quoting the new media reader from MIT press [Wardrip-Fruin , 2003]:

> *"The World-Wide Web (W3) was developed to be a pool of human knowledge, and human culture, which would allow collaborators in remote sites to share their ideas and all aspects of a common project."*

The last two decades have witnessed many significant attempts to make this knowledge "discoverable". These attempts broadly fall into two categories:

> (1) classification of webpages in hierarchical categories (directory structure), championed by the likes of Yahoo! and Open Directory Project;
> (2) full-text index search engines such as Excite, AltaVista, and Google.

The former is an intuitive method of arranging web pages, where subject-matter experts collect and annotate pages for each category, much like books are classified in a library. With the rapid growth of the web, however, the popularity of this method gradually declined. First, the strictly manual editorial process could not cope with the increase in the number of web pages. Second, the user's idea of what sub-tree(s) to seek for a particular topic was expected to be in line with the editors', who were responsible for the classification. We are most familiar with the latter approach today, which presents the user with a keyword search interface and uses a pre-computed web index to algorithmically retrieve and rank web pages that satisfy the query. In fact, this is probably the most widely used method for navigating through cyberspace. The earliest search engines had to handle orders of magnitude more documents than previous information retrieval systems. In fact, around 1995, when the number of static web pages was believed to double every few months, AltaVista reported having crawled and indexed approximately 25 million webpages. Indices of today's search engines are several orders of magnitude larger; Google reported around 25 billion web pages in 2005 [Patterson, 2005], while Cuil indexed 120 billion pages in 2008 [Arrington, 2008]. Harnessing together the power of hundreds, if not thousands, of machines has proven key in addressing this challenge of grand scale.

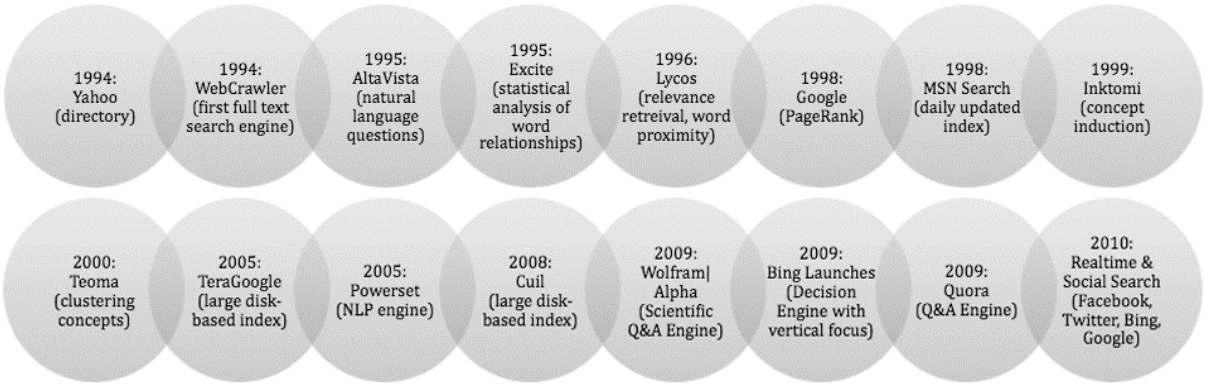

Figure 1: History of Major Web Search Engine Innovations (1994-2010)

Using search engines has become routine nowadays, but they too have followed an evolutionary path. Jerry Yang and David Filo created *Yahoo* in 1994, starting it out as a listing of their favorite web sites along with a description of each page [Yahoo, 2010]. Later in 1994, *WebCrawler* was introduced which was the first full-text search engine on the Internet; the entire text of each page was indexed for the first time. Introduced in 1993 by six Stanford University students, *Excite* became functional in December 1995. It used statistical analysis of word relationships to aid in the search process and is part of *AskJeeves* today. *Lycos*, created at CMU by Dr. Michael Mauldin, introduced relevance retrieval, prefix matching, and word proximity in 1994. Though it was the largest of any search engine at the time, indexing over 60 million documents in 1996, it ceased crawling the web for its own index in April 1999. Today it provides access to human-powered results from *LookSmart* for popular queries and crawler-based results from *Yahoo* for others. *Infoseek* went online in 1995 and is now owned by the Walt Disney Internet Group.

AltaVista, also started in 1995, was the first search engine to allow natural language questions and advanced searching techniques. It also provided multimedia search for photos, music, and videos. *Inktomi* was started in 1996 at UC Berkeley, and in June of 1999 introduced a directory search engine powered by *concept induction* technology. This technology tries to model human conceptual classification of content, and projects this intelligence across millions of documents. *Yahoo* purchased *Inktomi* was in 2003. *AskJeeves* launched in 1997 and became famous for being the natural language search engine, that allowed the user to search by framing queries in question form and responding with what seemed to be the right answer. In reality, behind the scenes, the company had many human editors who monitored search logs and located what seemed to be the best sites to match the most popular queries. In 1999, they acquired *Direct Hit*, which had developed the world's first click popularity search technology, and in 2001, they acquired *Teoma* whose index was built upon clustering concepts of subject-specific popularity. Google, developed by Sergey Brin and Larry Page at Stanford University, launched in 1998 and used inbound links to rank sites. The *MSN Search* and *Open Directory Project* were also started in 1998, of which the former reincarnated as *Bing* in 2009. The Open Directory, according to its website, "is the largest, most comprehensive human-edited directory of the Web". Formerly known as *NewHoo*, it was acquired by AOL Time Warner-owned Netscape in November 1998.

All current search engines rank web pages to identify potential answers to a query. Borrowing from information retrieval, a statistical similarity measure has always been used in practice to assess the closeness of each document (web page) to the user text (query); the underlying principle being that the higher the similarity score, the greater the estimated likelihood that it is relevant to the user. This similarity formulation is based on models of documents and queries, the most effective of which is the vector space model [Salton, 1975]. The cosine measure [Salton, 1962] has consistently been found to be the most successful similarity measure in using this model. It considers document properties as vectors, and takes as distance function the cosine of the angle between each vector pair. From an entropy-based

perspective, the score assigned to a document can be interpreted as the sum of information conveyed by query terms in the document. Intuitively, one would like to accumulate evidence by giving more weight to documents that match a query term several times as opposed to ones that contain it only once. Each term's contribution is weighted such that terms appearing to be discriminatory are favored while reducing the impact of more common terms. Most similarity measures are a composition of a few statistical values: frequency of a term t in a document d (term frequency or TF), frequency of a term t in a query, number of documents containing a term t (document frequency or DF), number of terms in a document, number of documents in the collection, and number of terms in the collection. Introduction of document-length pivoting [Singhal, 1996] addressed the issue of long documents either containing too many terms, or many instances of the same term.

The explosive growth of the web can primarily be attributed to the decentralization of content publication, with essentially no control of authorship. A huge drawback of this is that web pages are often a mix of facts, rumors, suppositions and even contradictions. In addition, web-page content that is trustworthy to one user may not be so to another. With search engines becoming the primary means to discover web content, however, users could no longer self-select sources they find trustworthy. Thus, a significant challenge for search engines is to assign a user-independent measure of trust to each website or webpage. Over time, search engines encountered another drawback [Manning, 2008] of web decentralization: the desire to manipulate webpage content for the purpose of appearing high up in search results. This is akin to companies using names that start with a long string of As to be listed early in the Yellow Pages. Content manipulation not only includes tricks like repeating multiple keywords in the same color as the background, but also sophisticated techniques such as cloaking and using doorway pages, which serve different content depending on whether the http request came from a crawler or a browser.

To combat such spammers, search engines started exploiting the connectivity graph, established by hyperlinks on web pages. Google [Brin, 1998] was the first web search engine known to apply link analysis on a large scale, although all web search engines currently make use of it. They assigned each page a score, called PageRank, which can be interpreted as the fraction of time that a random web surfer will spend on that webpage when following the out-links from each page on the web. Another interpretation is that when a page links to another page, it is effectively casting a vote of confidence. PageRank calculates a page's importance from the votes cast for it. HITS is another technique employing link analysis which scores pages as both hubs and authorities, where a good hub is one that links to many good authorities, and a good authority is one that is linked from many good hubs. It was developed by Jon Kleinberg and formed the basis of Teoma [Kleinberg, 1999].

Search engines aim not only to give quality results but also to produce these results as fast as possible. With several terabytes of data spread over billions of documents in thousands of computers, their systems are enormous in scale. In comparison, the text of all the books held in a small university might occupy only around 100 GB. In order to create such highly available systems, which continually index the growing web and serve queries with sub-second response times, it must optimize all resources: disk, memory, CPU time, as well as disk transfers.

This chapter describes how the index of a web-scale search engine organizes all the information contained in its documents. In the following sections, we will cover the basics of indexing data structures, introduce techniques that improve relevance of results, discuss trade-offs to best utilize the machine resources, and cover distributed processing concepts that allow scaling and updating of the index. In particular, we will delve into the topics of indexing phrases instead of terms, storage in memory vs. on disk, and data partitioning. We will conclude with some thoughts on information organization for the newly emerging data-forms.

ORGANIZING THE WEB

In order to avoid linearly scanning each webpage at query time, an *index* of all possible query terms is prepared in advance. Lets consider the collection of English books in a library. The simplest approach would be to keep track of all words from the English dictionary that appear in each book. On repeating this across all books, we end up with a *term-incidence matrix*, in which each entry tells us if a specific word occurs in a book or not. Figure 2 shows a sample term-incidence matrix. The collection of documents over which a search engine performs retrieval is referred to as a *corpus*. So, for a corpus of 1M documents with 100K distinct words, ~10GB (1M x 100K) will be required to hold the index in matrix form. The corpus itself will require around 4 bytes to encode each distinct word and hence a total of 4 GB (1M x 1000 x 4) storage if each document is 1000 words long on average. Clearly, lot of space is wasted in recording the absence of terms in a document, and hence a much better representation is to record only the occurrences.

term	document identifier					
	1	2	3	4	5	6
the	X	X	X	X	X	X
to	X		X	X	X	X
john		X		X		X
realize	X		X			X
algorithm					X	

Figure 2: Term-Incidence Matrix for a sample of English documents

The most efficient index structure is an *inverted index*: a collection of lists, one per *term*, recording the documents containing that term. Each item in the list for a term *t*, also referred to as a *posting*, records the ordinal document identifier *d*, and its corresponding term frequency (TF): <*d, tf*>. Note that if 4 bytes are used to encode each posting, a term appearing in ~100K documents will result in a posting list of size 100KB to 1MB; though most terms will have much shorter posting lists. We illustrate this in Figure 3 for the same example as in Figure 2.

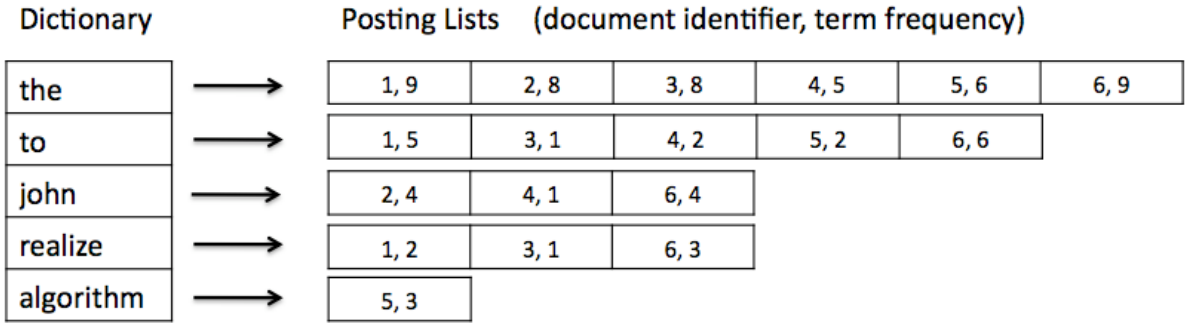

Figure 3: Illustration of Posting Lists for Example from Figure 2

In our example above, all terms in the English dictionary were known before hand. This, however, does not hold true on the web where authors create content in a multitude of languages, along with large

variations in grammar and style. Webpages are often found riddled with text in various colors and fonts, as well as images that lead to richer textual content when clicked, thereby providing no clear semantic structure. In addition, the character sequences are encoded using one of many byte-encoding schemes, such as UTF-8 or other vendor-specific standards. Since any visible component of a webpage might reasonably be used as query term, we take a superset of all spoken words. The set also includes numbers, constructs such as IA-32 or X-86, as well as tokens appearing in any URL. This collection of terms in an index is conventionally called a *dictionary* or *lexicon*. Dictionary and posting lists are the central data structures used in a search engine.

Building a Dictionary of Terms

For efficiency purposes, an identifier is used to represent each term in the dictionary, instead of storing them as strings. This mapping is either created on the fly while processing the entire corpus, or is created in two passes. The first pass compiles the dictionary while the second pass constructs the index. In both cases, the first step is to turn each document into a list of tokens and then use linguistic preprocessing to normalize them into indexing terms. This involves simple steps like breaking down sentences on whitespace and eliminating punctuation characters, as well as tricky steps like analyzing uses of the apostrophe for possession and verb contractions. Another common practice is *case-folding* by which all letters are reduced to lower case. This is a good idea in general since it allows the query *automobile* to also match instances of *Automobile* (which usually occurs at the beginning of a sentence). Another use case is in matching words with diacritics since users often enter queries without the diacritics. Documents also tend to use different forms of the same word, such as *realize*, *realizes*, and *realizing*. *Stemming* is a heuristic process that chops off the ends of words in the hope of collapsing derivationally related words. The most common algorithm used for stemming English words is *Porter's algorithm* [Porter, 1980]. Foreign languages require even more sophisticated techniques for term tokenization [Fung, 1998 and Chiang, 1992].

Certain terms, such as '*the*' and '*to*', are extremely common across documents and hence add little value towards matching specific documents to bag-of-words queries. All such terms can be identified by sorting the dictionary terms by their document frequency (DF), and then selecting the most frequent terms. Posting lists corresponding to such terms tend to be very long too, thereby adding to query processing cost. Removing these frequently occurring words (*stop* words) from the dictionary seems like a good idea since it does little harm and saves considerable storage space. Search engines, however, tend not to discard them since they play an important role in queries framed as phrases. By placing double quotes around a set of words, users ask to consider those words in precisely that order without any change. Eliminating 'the' or 'who' in a query like "The Who" will completely alter its meaning and user intent. Later in this chapter, we will discuss how compression techniques overcome the storage cost of posting lists for common words.

Answering The User's Query

Now we look at how retrieval is performed for a typical query using an inverted index. Given a query of three terms, the first step is to find those terms in the dictionary. Following that, the corresponding posting lists are fetched (and transferred to memory if residing on disk). Intersecting the lists on document identifiers then retrieves the relevant documents. A key insight is to start with the least frequent term since its posting list will be the shortest. Finally, the retrieved set of documents are ranked and re-ordered to present to the user. Given the small corpus size (1M), the above operations can be performed on any machine in well under a second. Understanding the usage of each computing resource is critical since search engines, built over thousands of machines, aim to not only give quality results but to produce these results as fast as possible.

Disk space is typically required to store the inverted posting lists;
Disk transfer is used to fetch inverted lists;
Memory is required for the dictionary and for accumulating documents from the fetched lists; and
CPU time is required for processing inverted lists and re-ordering them.

Performance optimizations of each of these components contribute towards several indexing design decisions. The choice of data structure for posting lists impacts both storage and CPU time. Search engines use both memory and disk to hold the various posting lists. If posting lists are kept in memory, a fixed-length array would be wasteful since common terms occur in many more documents (longer posting lists) compared to others. Singly linked lists and variable length arrays offer two good alternatives. While singly linked lists allow cheap updates such as insertion of documents following a new crawl, variable length arrays win in space requirement by avoiding the overhead for pointers. Variable length arrays also require less CPU time because of their use of contiguous memory, which in addition enables speedup through caching. A potential hybrid scheme is to use a linked list of fixed-length arrays for each term.

When storing posting lists on disk, it is better to store the postings contiguously without explicit pointers. This not only conserves space, but also requires only one disk seek to read most posting lists into memory. Lets consider an alternative in which lists are composed of a sequence of blocks that are linked in some way. Recall that there is a huge variance in size of posting lists; a typical term requires anywhere from 100KB to 1MB, a common term requires many times more, but most terms require less than 1KB for their lists. This places a severe constraint on the size of a fixed-size block, and significantly degrades typical query evaluation time. Apart from demanding additional space for next-block pointers, it also complicates update procedures.

Speeding Up Multi-Term Queries

As mentioned before, a typical query evaluation requires fetching multiple posting lists and intersecting them to quickly find documents that contain all terms. This *intersection* operation is a crucial one in determining query evaluation time. A simple and effective method is the merge algorithm: for a two word query, it maintains pointers into both lists and walks through them together by comparing the ordinal document identifiers. If they are the same, the document is selected and both pointers are advanced; otherwise the one pointing to the smaller identifier advances. Hence the operating time is linear in the size of posting lists, which in turn is bounded by the corpus size.

One way to process posting list intersection in sub-linear time is to use a *skip list* [Pugh, 1990], which augments a posting list with pointers that point to a document further down the list. Skip pointers are effectively shortcuts that allow us to avoid processing parts of the posting list that will not get intersected. Lets first understand how it allows efficient merging. Suppose we've stepped through two lists and both pointers have matched document 8 on each list. After advancing the pointers, list A points to 16 while list B points to 41. At this point we know that documents between 16 and 41 will have no effect on intersection. List A will consider the skip pointer at 16 and check if it skips to a document less than or equal to 41; if it doesn't, following the skip pointer avoids all those comparisons with list B's 41. As more skips are made, processing gets even faster.

A number of variant versions of posting list intersection with skip pointers is possible depending on when exactly the skip pointer is checked [Moffat, 1996]. Deciding where to place skip pointers is a bit tricky. More skips imply shorter skip spans, and hence more opportunities to skip. But this also means lots of comparisons to skip pointers, and lots of space for storing skip pointers. On the other hand, though fewer skips require less pointer comparisons, the resulting longer skip spans provide fewer opportunities to skip. A simple heuristic, which has been found to work well in practice, is to use $\sqrt{P}$ evenly spaced skip pointers for a posting list of length P.

Better Understanding of User Intent

When ranking multi-term queries, one of the prominent signals used is the proximity of different terms on a page. The goal is to prefer documents in which query terms appear closest together over the ones in which they are spread apart. Proximity of terms is even more critical in the case of phrase queries, where relative position of each query term matters. Rather than simply checking if terms are present in a document, we also need to check that their positions of appearance in the document are compatible with the phrase query being evaluated. This requires working out offsets between words. Posting lists typically add word positions to index entries so that the locations of terms in documents can be checked during query evaluation.

Creating a positional index significantly expands storage requirements. It also slows down query processing since only a tiny fraction of the documents that contain the query terms also contain them as a phrase, thereby needing to skip over the positional information in each non-matched posting. This also results in processing cost often being dominated by common terms since they occur at the start or in the middle of any phrase.

In order to better represent an author's intent in a document and better match a user's intent in their query, there has been significant work done in phrase-based indexing. However, such indexing is potentially expensive. There is no obvious mechanism for accurately identifying which phrases might be used in queries, and the number of candidate phrases is enormous since they grow far more rapidly than the number of distinct terms. For instance, if a dictionary has 200,000 unique terms, and we consider all 1-5 word phrases, the phrase dictionary will be of size greater than 3.2*10^26 – much larger than any existing system can store in memory or manipulate. In order to have a manageable dictionary, only "good" phrases are indexed. A good phrase has terms that often appear together or appear in delimited sections (e.g. titles and headings). Eliminating phrases that are subphrases of longer phrases also helps trim the list. Some phrase-based indexers also keep track of phrases that often appear together in order to generate a related-phrases list. This enables a page that mentions "Rat Terrier" to return for a query of "Charlie Feist"[1]. For phrases composed of rare words, having a phrase index yields little advantage, as processing savings are offset by the need to access a much larger dictionary. A successful strategy is to have an index for word pairs that begin with a common word and combine it with a word-level inverted index.

In practice, Google (through the TeraGoogle project [Patterson, 2004]), Yahoo! (through the use of 'superunits' [Kapur, 2004]) and startups such as Cuil have experimented with phrase-based indexing. Table 1 below summarizes the advantages and disadvantages of Term- and Phrase-based Indexing.

	Advantages	Disadvantages
Term-Based Indexing	Limited number of posting lists	Average posting list size is longer
	Simple to implement	Storing positional information bloats index size
		Calculating proximity of multiple terms can be an expensive operation
Phrase-Based Indexing	Better measure of intent	Larger dictionary: More posting lists to manage
	Ability to index related phrases	Breaking user query into phrases correctly is difficult

Table 1: Term- vs. Phrase-based Indexing

[1] A feist is a type of small hunting dog, developed in the rural southern United States.

LAYING OUT THE INDEX

In order to handle the load of a modern search engine, a combination of distribution and replication techniques is required. Distribution refers to the fact that the document collection and its index are split across multiple machines and that answers to the query as a whole must be synthesized from the various collection components. Replication (or mirroring) then involves making enough identical copies of the system so that the required query load can be handled even during single or multiple machine failures. In this section we discuss the various considerations for optimally dividing data across a distributed system in the most optimal way.

The decision of how to divide the data includes a balancing act between the number of posting lists, the size of each posting list and the penalties involved when multiple posting lists need to accessed at the same machine. Indexing phrases implies having many more (and possibly shorter) posting lists as the number of terms is significantly fewer than the number of phrases. For this section, we will assume the following are constant:

1. Number of phrases in our dictionary
2. Number of documents to be indexed across the system
3. Number of machines

Document vs. Term Based Partitioning

There are two common ways of distributing index data across a cluster: by document or by term, as illustrated here in Figure 4 for the same example as before. Let's discuss the design of each system before examining the advantages and disadvantages of each.

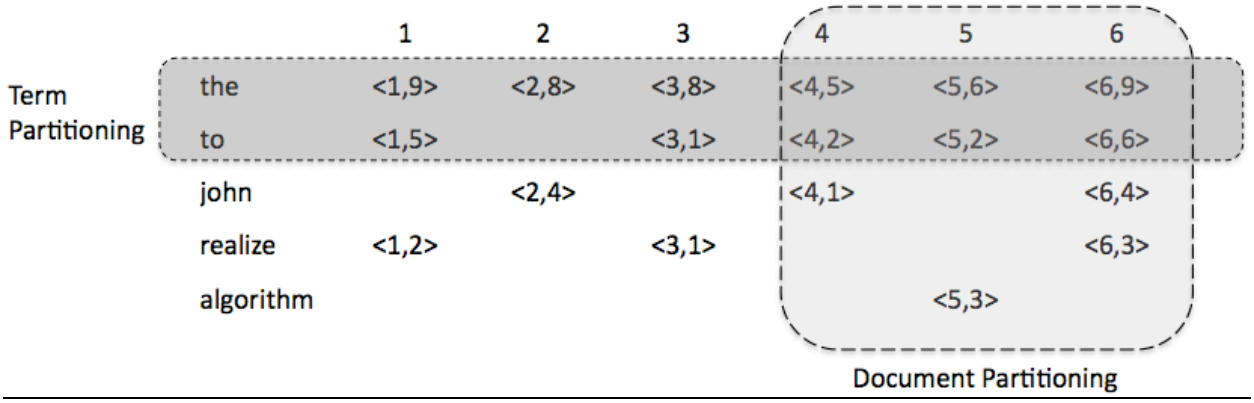

Figure 4: Illustrating Document- vs. Phrase-Based Partitioning for Posting Lists shown in Figure 3

Document Based Partitioning

The simplest distribution regime is to partition the collection and allocate one sub-collection to each of the processors. A local index is built for each sub-collection; when queries arrive, they are passed to every sub-collection and evaluated against every local index. The sets of sub-collection answers are then combined in some way to provide an overall set of answers. An index partitioned by document saves all information relevant to that document on a single machine. The number of posting lists on a machine is thus dependent on the number of unique terms or phrases that appear in the corpus of documents indexed on the given machine. Also, the upper limit on the size of each posting list is the number of documents indexed on the machine. In the example above, we have partitioned the document-space into two indices: the first index contains all information about documents 1, 2 and 3 while the second index contains all information about documents 4, 5 and 6.

The benefit of storing all the posting lists for a given document on a single machine is that intersections, if needed, can be performed locally. For each query, a master index-server dispatches the query to all the workers under it. Each worker server looks up the query (or intersection) on its local index and returns results to the master. In effect, each worker is an independent search engine for the pages that have been indexed and stored in its memory. Efficient implementations have a fan-out between 70 and 80. As the size of the index grows, a hierarchical structure must be created. However, given the expectation of sub-second load times from search engines, the depth of the lookup tree rarely exceeds two.

A document-partitioned index allows for index construction and document insertion more naturally. One of the hosts can be designated to have a dynamic corpus so that it is the only one to rebuild its index. It also allows the search service to be provided even when one or more of the hosts are offline. Another advantage of a document-partitioned index is that the computationally expensive parts of the process are distributed equally across all of the hosts in the computer cluster.

Term Based Partitioning

The alternative to document based partitioning is term or phrase based partitioning. In a term-partitioned index, the index is split into components by partitioning the dictionary. Each processor has full information about only a subset of the terms. This implies that to handle a query, only the relevant subset of processors needs to respond. Since all postings for a given term are stored on a single machine, posting lists in a term-partitioned index are often significantly longer than their document-partitioned counterparts. Also, unlike the document-partitioned index case, posting lists for different terms that are in the same document can be stored on different machines. Furthermore, each machine can have a different number of posting lists stored on it (depending on machine limits and relative posting list sizes). In the example in Figure 4, we have partitioned the term-space into two indices: the first index contains all information about the terms '*the*' and '*to*' while the second index contains all information about the terms '*john*', '*realize*' and '*algorithm*'.

Since all postings for a given term are stored in one place in a cluster, not all servers need to do work for each query. The master index-server only contacts the relevant worker server(s). This method requires fewer disk seeks and transfer operations during query evaluation than a document-partitioned index because each term's inverted list is stored contiguously on a single machine rather than in fragments across multiple machines. If a cluster is managed efficiently, it can retrieve results to multiple query requests at the same time.

When comparing this index architecture to its document-partitioned counterpart, we observe a few drawbacks. Firstly, intersections cannot always take place locally. If the terms being intersected reside on different machines, one of the posting lists needs to be copied over to the other machine in order to do a local intersection. Usually, the shorter posting list is copied to reduce network traffic. Secondly, the disk transfer operations involve large amounts of data since posting lists are longer. The coordinating machine can easily get overloaded and become a bottleneck, thereby starving the other processors of work. Table 2, below, summarizes the tradeoffs.

	Advantages	Disadvantages
Document-Partitioned Index	Posting Lists are Smaller	All machines are contacted for each query
	Easy to manage and scale	Merging results from all machines can be expensive
	Intersections are local to a machine	Network traffic to all machines can be expensive
	Index updates can be done dynamically and one partition at a time	
Term-Partitioned Index	Only a subset of machines are used for each query	Remote intersections can be expensive as entire posting lists need to be copied over network
		Index updates need to be done for entire index

Table 2: Document- vs. Term-Based Partitioning

Memory vs. Disk Storage

The next decision we have to make is deciding whether to store the indexing components in memory or on disk. It is tempting to store the dictionary in memory because doing so means that a disk access is avoided for every query term. However, if it is large, keeping it in memory reduces the space available for caching of other information and may not be beneficial overall. Fortunately, access to the dictionary is only a small component of query processing; if a B-tree-like structure is used with the leaf nodes on disk and internal nodes in memory, then a term's information can be accessed using just a single disk access, and only a relatively small amount of main memory is permanently consumed. Lastly, recently or frequently accessed query terms can be cached in a separate small table.

Memory Based Index

The two major search engines today, Google and Bing, store the majority of their search engine indices in memory. The benefit of this architecture is that the local lookup time is almost instantaneous. The drawback is that memory is extremely expensive and doesn't scale infinitely. As the size of the web increases, the size of the index increases linearly as does the number of machines needed to hold a copy of this index. At serving time, these search engines employ a scatter-gather approach to find the best results. A master index-server dispatches a query lookup to all the worker machines, and waits for results. The index is usually document-partitioned and each mini-search-engine looks up the relevant results and returns them to the master index-server. As they arrive, the master does a merge-sort of the results received and returns the final result. Since the index is held in memory, the majority of time in this architecture is spent in the data transfers across the network.

While we can imagine an index that is term-partitioned and based in memory, there have been no major search engines that have taken such an approach. This is primarily because posting lists of many terms on the web are too large to fit in memory.

Disk Based Index

In the last few years, Google (through the TeraGoogle project [Patterson, 2004]), and startups such as Cuil have experimented with disk-based indices wherein data is organized in a way that requires fewer lookups and very few network transfers for each query. Both of these indices have been term-partitioned. In these architectures, the time is spent in disk-seeks and block reads rather than in network transfers and

merge-sorts. It is possible to have a disk-based and document partitioned index. No search engine in the recent past has built one with this configuration because each lookup would involve multiple disk seeks on each machine in the cluster.

The big advantage of a disk-based index is that it can scale more cost-efficiently than its memory-based counterpart. Disk is ~100x cheaper than memory while providing significantly larger amounts of storage space. As the number of documents on the web increases, posting lists will get longer in the disk-based index. The increased penalty of traversing longer posting lists is negligible. On the other hand, in the memory-based index, the number of machines needed to support the index increases with the size of the index. Hence, query processing time is limited by the per-machine lookup time. Once the system starts slowing down (upper limit on performance), the only way to scale further is to duplicate the index across multiple copies of the index and replicate entire clusters. Table 3 summarizes the advantages and disadvantages of these options.

	Advantages	Disadvantages
Memory-based Index	Extremely fast lookup	Memory is expensive
	Works well in document-partitioned case	Number of machines needed grows as fast as size of index
Disk-based Index	Disk is cheap	Disk is slow so number of lookups need to be minimized
	Longer posting lists can be stored contiguously	Not ideal in document-partitioned case because it involves too many disk seeks

Table 3: Memory- vs. Disk-Based Indices

Compressing The Index

From the discussion so far, it is clear that a web-scale index makes storage space a premium resource for search engines. An attractive solution for conserving space is to use a highly compressed inverted index. Decompression at query evaluation time, however, made it an expensive proposition in the past since CPUs were slower. This trend has reversed and decompression algorithms on modern hardware run so fast that the cost of transferring a compressed chunk of data from disk and then decompressing it is usually far less expensive than that of transferring the same chunk of data in uncompressed form. As the ratio of processor speed to disk speed continues to diverge, reducing posting list sizes promises increasingly more performance gains.

Using shorter posting lists has more subtle benefits for a disk-based index. First, it makes it faster to transfer data from disk to memory. More importantly, it reduces disk seek times since the index is smaller. These reductions more than offset the cost of decompressing, thereby reducing the overall query evaluation time. Another beneficial outcome of compression is the increased use of caching. Typically, web queries come with a skewed distribution where certain query terms are more common than others. If the posting list for a frequently used query term is cached, all queries involving that term can be processed entirely in memory and not involve any disk seek. Even for a memory-based index, the cost of decompressing is more than offset by the reduction in memory-to-cache transfers of larger uncompressed data. Since memory is a more expensive resource than disk space, increased speed due to caching has proved to be the primary motivator for using compression in today's search engines [Zhang, 2007].

In the rest of this section, we will discuss simple compression schemes that can not only keep the penalty

of decompressing a posting list small, but also cut the storage cost of an inverted index by almost 75%. We begin with the observation that document identifiers for frequent terms are close together. When going over documents one by one, we will easily find terms like ‘the’ and ‘to’ in every document, but to search for a term like ‘john’ we might have to skip a few documents every now and then. The key insight here is that *gaps* between document identifiers in postings are short, requiring a lot less space to encode than say the 20 bits needed in a 1M corpus for document identifier. This is illustrated below using our earlier example. Rarer terms, however, occur only once or twice in a collection and hence their gaps will have the same order of magnitude as the document identifiers. We will need a *variable encoding* representation that uses fewer bits for short gaps but does not reduce the maximum magnitude of a gap.

Original posting lists:
the: ⟨1, 9⟩ ⟨2, 8⟩ ⟨3, 8⟩ ⟨4, 5⟩ ⟨5, 6⟩ ⟨6, 9⟩
to: ⟨1, 5⟩ ⟨3, 1⟩ ⟨4, 2⟩ ⟨5, 2⟩ ⟨6, 6⟩
john: ⟨2, 4⟩ ⟨4, 1⟩ ⟨6, 4⟩

With gaps:
the: ⟨1, 9⟩ ⟨1, 8⟩ ⟨1, 8⟩ ⟨1, 5⟩ ⟨1, 6⟩ ⟨1, 9⟩
to: ⟨1, 5⟩ ⟨2, 1⟩ ⟨1, 2⟩ ⟨1, 2⟩ ⟨1, 6⟩
john: ⟨2, 4⟩ ⟨2, 1⟩ ⟨2, 4⟩

Variable byte (VB) encoding [Witten, 1999] uses an integral but adaptive number of bytes depending on the size of a gap. The first bit of each byte is a *continuation bit*, which is flipped only in the last byte of the encoded gap. The remaining 7 bits in each byte are used to encode part of the gap. To decode a variable byte code, we read a sequence of bytes until the continuation bit flips. We then extract and concatenate the 7-bit parts to get the magnitude of a gap. Since it reduces the average magnitude of all gaps in a posting list, and is simple to implement, compression techniques benefit greatly from such a transformation. The idea of VB encoding can also be applied to larger or smaller units than bytes, such as 32-bit words and 4-bit *nibbles*. Larger words decrease the amount of bit manipulation necessary at the cost of less effective (or no) compression. Units smaller than bytes achieve even better compression ratios but at the cost of more bit manipulation. In general, variable byte codes offer a good compromise between compression ratio (space) and speed of decompression (time).

If disk space is at a premium, we can get even better compression ratios by using bit-level encoding [Golomb, 1966]. These codes, in particular the closely related *g (gamma)* and *d (delta)* codes [Elias, 1975, Rice 1979], adapt the length of the code on a finer grained bit level. Each codeword has two parts, a prefix and a suffix. The prefix indicates the binary magnitude of the value and tells the decoder how many bits there are in the suffix part. The suffix indicates the value of the number within the corresponding binary range. Inspite of greater compression ratios, these codes are expensive to decode in practice. This is primarily because code boundaries usually lie somewhere in the middle of a machine word, making it necessary to use bit-level operations such as shifts and masks for decoding. As a result, query processing is more time consuming for *g* and *d* codes than for variable byte codes.

The choice of coding scheme also affects total fetch-and-decode times, where the byte-wise and word-aligned codes enjoy a clear advantage. [Scholer, 2002] found that variable byte codes process queries twice as fast as either bit-level compressed indexes or uncompressed indexes, but pay for a 30% penalty in the compression ratio when compared with the best bit-level compression method. [Trotman, 2003] recommended using VB codes unless disk space is a highly scarce resource. Both studies also show that compressed indexes are superior to uncompressed indexes in disk usage. In a later study [Anh, 2005], variable nibble codes showed 5% to 10% better compression and upto one-third worse effectiveness, in comparison to VB codes. These studies clearly demonstrate that using simple and efficient decompression methods substantially decreases the response time of a system. Use of codes does, however, present

problems for index updates. Since it involves decoding the existing list and recoding with new parameters, processing the existing list becomes the dominant cost of the update.

The effectiveness of compression regimes is particularly evident in posting lists for common words, which require only a few bits per posting after compression. Let's consider stop words, for instance. Since these words are likely to occur in almost every document, a vast majority of gaps in their postings can be represented in just a bit or two. Allowing for the corresponding term frequency (TF) value to be stored in at most 10-11 bits (TF of upto ~1000), each posting requires a total of only 12 bits. This is almost a quarter of ~40 bits that would be required if the postings were stored uncompressed. Thus, even though maintaining a list of stop words seemed like an attractive proposition when the index was uncompressed, the additional space savings do not carry over to the size of the compressed index. And even though less frequent terms require longer codes for their gaps, their postings get encoded in a few bits on average since they require shorter codes for their TF values.

For a positional index, word positions account for a bulk of the size in uncompressed form. For instance, it takes almost two bytes to ensure that all positions can be encoded in a document of upto 64K words. This cost can be significantly reduced by representing only the difference in positions, just like we did for document identifiers. These gaps can either be localized to within each document, or can be global across all documents [Zobel, 2006]. In the latter case, two sets of codes are used: one that represents the document gaps, and a second to code the position intervals between appearances of a term. While byte-aligned encoding can be used to quickly decode the document identifiers, more efficient interleaved bit-level encoding can be used for the positions. They are decoded in parallel when both components are required during query evaluation. As discussed above, for common words compression ratios of 1:4 are easy to achieve without positional information. In a positional index, however, the average per document requirement for common words is much larger because of the comparatively large number of word-gap codes that must be stored.

The compression techniques we describe are *lossless*, that is, all information is preserved. Better compression ratios can be achieved with *lossy compression*, which discards some information. Case folding, stemming, and stop word elimination are forms of lossy compression. Similarly, dimensionality reduction techniques like latent semantic indexing create compact presentations from which we cannot fully restore the original collection. Lossy compression makes sense when the "lost" information is unlikely ever to be used by the search system, for instance postings far down the list in an impact-sorted index (described next) can be discarded.

Ordering by Highest Impact First

To reduce disk transfer costs, it is necessary to avoid fetching the posting lists in their entirety. This is particularly true for common terms for which posting lists are very long. An attractive option is to rearrange the list itself so that only a part of each relevant list is fetched for a typical query. For instance, if only the largest term frequency values of a posting list contribute to anything useful, it makes sense to store them at the beginning of the list rather than somewhere in the middle of the document-based ordering. Ordering by term frequency also allows for the scanning of many posting lists to be terminated early because smaller term weights do not change the ranking of the highest ranked documents.

Using a frequency-ordered index, a simple query evaluation algorithm would be to fetch each list in turn and process only those values (TF x IDF) that contribute to a threshold S or higher. If disk reads are performed one block at a time, rather than on the basis of entire posting list, this strategy significantly reduces disk traffic without degrading effectiveness. A practical alternative is to use only the first disk block of each list to hold the high impact postings; the remainder of the list can stay sorted in document order. These important first blocks could then all be processed before the remainder of any lists, thereby

ensuring that all terms are able to contribute relevant documents. One could also interleave their respective processing; once the first block of each list has been fetched and is available in memory, the list with the highest posting value is selected and its first run of pointers are processed. Attention then switches to the list with the next-highest run, which could either be in a different list or in the same list. So, each list is visited one or more times, depending on the perceived contribution of that term to the query. Since the most significant index information is processed first, query evaluation can be terminated by a time bound rather than a threshold.

So how does this new ordering affect storage requirements? Since the inverted lists are read in blocks rather than in their entirety, contiguous storage is no longer a necessity. Blocks with high-impact information could be clustered on disk, further accelerating query processing. The long lists for common terms will never be fully read, saving a great deal of disk traffic. Query evaluation becomes a matter of processing as many blocks as can be handled within the time that is available. Let's look at an example posting list from [Zobel, 2006] now to understand the impact on compression, shown here as document ordered (<doc id, term frequency>):
⟨12, 2⟩ ⟨17, 2⟩ ⟨29, 1⟩ ⟨32, 1⟩ ⟨40, 6⟩ ⟨78, 1⟩ ⟨101, 3⟩ ⟨106, 1⟩.
When the list is reordered by term frequency, it gets transformed:
⟨40, 6⟩ ⟨101, 3⟩ ⟨12, 2⟩ ⟨17, 2⟩ ⟨29, 1⟩ ⟨32, 1⟩ ⟨78, 1⟩ ⟨106, 1⟩.
The repeated frequency information can then be factored out into a prefix component with a counter that indicates how many documents there are with this same frequency value:
⟨6 : 1 : 40⟩ ⟨3 : 1 : 101⟩ ⟨2 : 2 : 12, 17⟩ ⟨1 : 4 : 29, 32, 78, 106⟩.
Not storing the repeated frequencies gives a considerable saving. Finally, if differences of document identifiers are taken, we get the following:
⟨6 : 1 : 40⟩ ⟨3 : 1 : 101⟩ ⟨2 : 2 : 12, 5⟩ ⟨1 : 4 : 29, 3, 46, 28⟩.
The document gaps within each equal-frequency segment of the list are now on average larger than when the document identifiers were sorted, thereby requiring more encoding bits/bytes. However, in combination with not storing repeated frequencies, these lists tend to be slightly smaller than document-sorted lists. The disadvantage, however, is that index updates are now more complex.

Since it made sense to order the posting lists by decreasing term frequency, it makes even more sense to order them by their actual impact. Then all that remains is to multiply each posting value by the respective query term weight, and then rank the documents. Storing pre-computed floating-point document scores is not a good idea, however, since they cannot be compressed as well as integers. Also, unlike repeated frequencies, we can no longer cluster exact scores together. In order to retain compression, the impact scores are quantized instead, storing one of a small number of distinct values in the index. The compressed size is still slightly larger compared to document- and frequency-sorted indexes because the average document gaps are bigger.

Managing Multiple Indices

Webpages are created and refreshed at different rates. Therefore, there is no reason to crawl and index pages uniformly. Some pages are inherently ever-changing (e.g. www.cnn.com) while others won't change for years (e.g. my grandmother's static homepage that I designed as a birthday gift 10 years ago). If a search system can learn, over a period of time, the rate of refreshing of a page, it can crawl and index the page only at the optimal rate.

The way we have described search indices so far makes a huge assumption: there will be a single unified index of the entire web. If this assumption was to be held, every single time we re-crawled and re-indexed a small set of fast-changing pages, we would have to re-compress every posting list for the web and push out a new web index. Re-compressing the entire index is not only time consuming, it is

downright wasteful. Why can we not have multiple indices -- bucketed by rate of refreshing? We can and that is what is standard industry practice. Three commonly used buckets are:

1. The large, rarely-refreshing pages index
2. The small, ever-refreshing pages index
3. The dynamic real-time/news pages index

At query-time, we do three parallel index lookups and merge the results based on the signals that are retrieved. It is common for the large index to be re-crawled and re-indexed as slow as every month, while the smaller index is refreshed daily, if not weekly. The dynamic, real time/news index is updated on a per-second basis.

Another feature that can be built into such a multi-tiered index structure is a waterfall approach. Pages discovered in one tier can be passed down to the next tier over time. Pages and domains can be moved from the rarely refreshing index to the ever-refreshing index and vice versa as the characteristics of pages change over time. Having such a modular and dynamic system is almost necessary to maintain an up-to-date index of the web.

As pages are re-crawled or re-indexed, older index and crawl file entries can be invalidated. Invalidations are often stored in a bit vector. As a page is re-indexed, the appropriate posting lists must be appropriately updated. There are many ways to update the posting lists but, in general, the per-list updates should be deferred for as long as possible to minimize the number of times each list is accessed. The simplest approach is to process as for the merging strategy and, when a memory limit is reached, then proceed through the whole index, amending each list in turn. Other possibilities are to update a list only when it is fetched in response to a query or to employ a background process that slowly cycles through the in-memory index, continuously updating entries. In practice, these methods are not as efficient as intermittent merge, which processes data on disk sequentially.

Merging is the most efficient strategy for update but has the drawback of requiring significant disk overheads. It allows relatively simple recovery as reconstruction requires only a copy of the index and the new documents. In contrast, incremental update proceeds in place with some space lost due to fragmentation. But recovery in an incremental index may be complex due to the need to track which inverted lists have been modified [Motzkin, 1994].

SCALING THE SYSTEM

Over the last decade, search engines have gone from crawling tens of millions of documents to over a trillion documents [Alpert, 2008]. Building an index of this scale can be a daunting task. On top of that serving hundreds of queries per second against such an index only makes the task harder. In mid 2004, the Google search engine processed more than 200 million queries a day against more than 20TB of crawled data, using more than 20,000 computers [Computer School, 2010].

Web search engines use *distributed indexing* algorithms for index construction because the data collections are too large to efficiently processed on a single machine. The result of the construction process is a distributed index that is partitioned across several machines. As discussed in the previous section this distributed index can be partitioned according to term or according to document.

While having the right infrastructure is not necessary (and search engines before 2004 did not have many of these pieces), they definitely can reduce the pain involved in building such a system. As the dependence on parallel processing increased, the need for an efficient parallel programming paradigm arose: a framework that provided the highest computational efficiency while maximizing programming efficiency [Asanovic, 2008]. In this section, we will summarize the features of a distributed file system and a map-shuffle-reduce system used within a web-indexer.

Distributed File System

A search engine is built from a copy of the web: a copy that is stored locally. If we only care to build a search engine of all the text documents on the web, we need to store this data on hundreds, if not thousands of machines. Assuming that the average webpage is 30k, and that there are 100 billion index-worthy billion webpages (we don't store and index all because many of them are junk or spam), we need 3 petabytes of storage just to store the pages. We know that the index is at least as big as the initial data, so we need at least 6 petabytes of storage for our system. If we load each server with 12 1-terabyte disks, we need 500 such servers to hold one copy of the web and its index – more if we want to build in some level of redundancy.

Search engines are built on top of commodity hardware. In order to manage such large amounts of data across large commodity clusters, a distributed file system that provides efficient remote file access, file transfers, and the ability to carry out concurrent independent operations while being extremely fault tolerant is essential [Silberschatz, 1994]. While there are many open source distributed file systems such as [MooseFS] and [GlusterFS], search engine companies such as Google have built their own proprietary distributed file systems [Ghemawat, 2003]. The main motivation for a company such as Google to develop its own file system is to optimize the operations that are most used in the web search indexing domain. One area where a web indexer differs from a traditional file system user is in the file access pattern. Traditionally, users or systems access files according to the 90/10 law [Smith, 1985] and caching provides huge performance gains on re-accesses. A web indexer streams through the data once while writing to multiple files in a somewhat random manner. This implies that if a file system optimized for web search indexing can provide efficient random writes (the common operation), it does not need to provide extremely efficient caching.

Once we have an efficient distributed file system, we can develop an indexer that can process data across a cluster and create a single index representing all the underlying data. Many of the indexer functions are data parallel, that is, the same code is run on independent sets of data. However, there are parts of the process that require related pieces of information from all the machines to be combined to create some of the signals (e.g. number of in-links to a given page). We find that there are three programming constructs that are used repeatedly in distributed computing such as building a web index. The next subsection describes such a Map-Shuffle-Reduce framework.

Map-Shuffle-Reduce

While the constructs of map and reduce have been around since the early days of Common Lisp and the idea of shuffling data around a cluster has been done since the beginning of distributed computing, Google was the first company to formally describe a framework [Dean, 2004] that did all this while providing fault tolerance and ease of use. In light of the plethora of literature defining this framework, we present an extremely short treatment of the definitions and focus on the uses in web-indexing.

Map: The master node chops up the problem into small chunks and assigns each chunk to a worker. The worker either processes the chunk of data with the mapper and returns the result to the master or further chops up the input data and assigns it hierarchically. Mappers output key-value pairs. E.g. when extracting anchors from a webpage, the mapper might output <dst_link, src_link> where dst_link is the link found on src_link.

Shuffle: This step is optional. In this step, data is transferred between nodes in order to group key-value pairs from the mapper output to in a way that enables proper reducing. E.g. in extracting anchors, we shuffle the output of the anchor extracting mapper so that all the anchors for a given link end up on the same machine.

Reduce: The master takes the sub-answers and combines them to create the final output. E.g. for the anchor extractor, the reducer re-organizes local data to have all the anchors for a given link be contiguous and outputs them along with a link that summarizes the findings (number of off-domain anchors, number of on-domain anchors, etc). Additionally, we can output a separate file that has just the summary line and also an offset into the anchor file so that we can seek to it and traverse all the anchors to a given link on demand.

There are many added features and benefits built into such a system. It hides the complexity of a parallel system from the programmer and scales with the size of the problem and available resources. Also, in the case of a crash, recovery is simple: the master simply reschedules the crashed worker's job to another worker.

That said, such a system is not required for many parts of the indexing system. The use of the map-shuffle-reduce framework can be avoided for tasks that have a trivial or non-existent shuffle step. For those tasks, a regular program will perform just as well (given that there is a fault tolerant infrastructure available). It is, in particular, not required for the calculation of signals such as PageRank (iterative graph traversal / matrix multiplication) or document-partitioned indexing (all the data for a given document is available locally and no shuffle is needed). On the other hand, to construct a term-partitioned index, shuffling is a key step. The posting lists from all the worker nodes corresponding to the same term need to be combined into a single posting list.

EXTRACTING FEATURES FOR RANKING

Any modern search engine index spans tens or hundreds of billions of pages, and most queries return hundreds of thousands of results. Since the user cannot parse so many documents, it is a pre-requisite to rank results from most relevant to least relevant. Earlier, we discussed using term frequency as an evidence (or signal) to give more weight to pages that have a term occurring several times over pages containing it only once. Other than such statistical measures, search engines extract several other signals or features from a page to understand the author's true intention, as well as key terms. These signals are then embedded in each posting for a term.

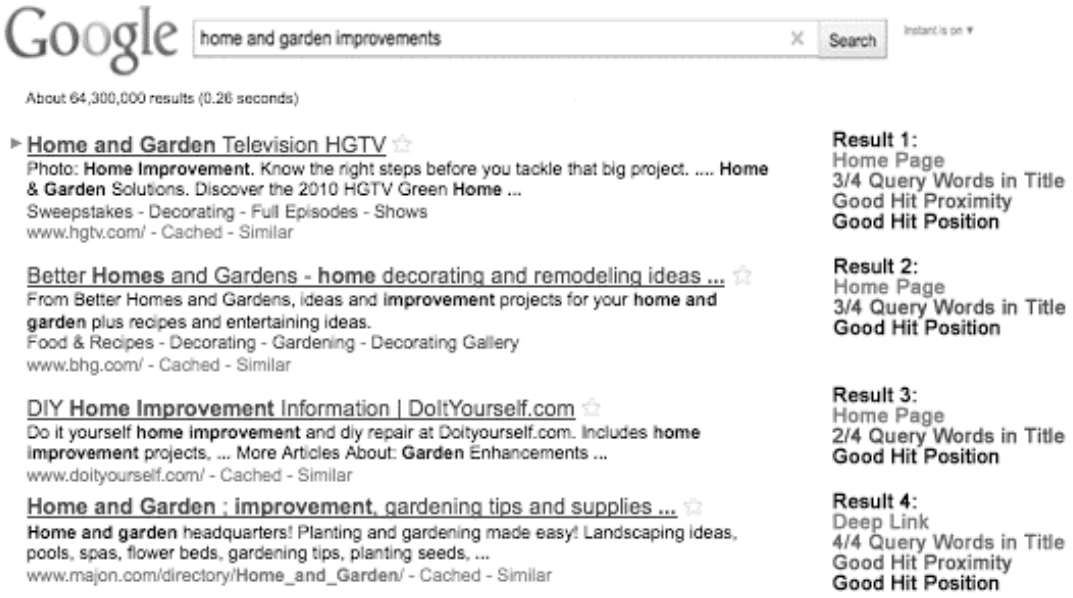

Figure 5: Importance of Basic On-Page Signals on Search Results

We will highlight a few of these features using results for 'home and garden improvements' on Google, as illustrated in Figure 5 above. First note that page structures, such as titles and headings, and url depth play a major role. Next we see that most terms occur close to each other in the results, highlighting the need for term positions or phrases during indexing. Also important, though not clear from the figure, is the respective position of terms on pages; users prefer pages that contain terms higher up in the page. Other than these, search engines also learn from patterns across the web and analyze pages for undesirable properties, such as presence of offensive terms, lots of outgoing links, or even bad sentence- or page-structures. The diversity and size of the web also enables systems to determine statistical features such as the average length of a good sentence, ratio of number of outgoing links to number of words on page, ratio of visible keywords to those not visible (meta tags or alt text), etc. Recent search start-ups such as PowerSet, Cuil and Blekko have attempted to process a page and extract more than term occurances on a page. PowerSet built an engine that extracted meaning out of sentences and could therefore be made part of a larger question and answer service. Cuil extracted clusters of phrases from each page to evaluate the topics that any page talks about. Blekko extracts entities such as time and locations in order to allow the user to 'slash', or filter, results by location, time or other user-defined slashes.

While in a perfect world indexing an author's intent in the form of on-page analysis should be good enough to return good results, there are too many search engine 'bombers' who stuff pages with keywords to fool an engine. In fact, most basic search engine optimization (SEO) firms focus on these on-page features in order to improve rankings for their sites. Hence, off-page signals have increasingly proved to be the difference between a good search engine and a not-so-good one. They allow search engines to determine what other pages say about a given page (anchor text) and whether the linking page itself is reputable (PageRank or HITS). These signals usually require large distributed platforms such as map-shuffle-reduce because they collect the aggregated information about a given page or domain as presented by the rest of the web. The final ranking is thus a blend of static *a priori* ordering that indicates if a page is relevant to queries in general, and a dynamic score which represents the probability that a page is relevant to the current query.

In PageRank, each page is assigned a score that simulates the actions of a random web surfer, who with probability p is equally likely to follow one of the links out of the current page, or with probability $1 - p$ chooses any other random page to move to. An iterative computation can be used to compute the long-term probability that such a user is visiting any particular page and that probability is then used to set the PageRank. The effect is that pages with many in-links tend to be assigned high PageRank values, especially if the source pages themselves have a high PageRank. On the other hand, pages with low in-link counts, or in- links from only relatively improbable pages, are considered to not hold much authority in answering queries. HITS, as described earlier, scores pages as both hubs and authorities, where a good hub is one that links to many good authorities, and a good authority is one that is linked from many good hubs. Essentially, hubs are useful information aggregations and provide broad categorization of topics, while authorities provide detailed information on a narrower facet of a topic. This is also computed iteratively using matrix computation based on the connectivity graph. However, instead of pre-computing the hub and authority scores at indexing time, each page is assigned a query specific score at serving time.

Naturally, spammers nowadays invest considerable time and effort in faking PageRank or Hubs and Authorities - this is called link spam. Because spammers will build clusters of webpages that link to each other in an effort to create the illusion of pages having good structure and good anchor text coming from 'other' pages, it is important to have good coverage of pages on the index and a good set of trusted seed pages. Moreover, they filter out links from known link farms and even penalize sites with links to such farms. They rightly figure that webmasters cannot control which sites link to their sites, but they can control which sites they link out to. For this reason, links into a site cannot harm the site, but links from a

site can be harmful if they link to penalized sites. To counter this, sites have focused on exchanging, buying, and selling links, often on a massive scale. Over the years there have been many such SEO companies that have tried to aggressively manipulate the effect of optimizations and gaming search engine ranking. Google and other search engines have, on some occasions, banned SEOs and their clients [Kesmodel, 2005 and Cutts, 2006] for being too aggressive.

FUTURE RESEARCH DIRECTIONS

While core web-search technology has some interesting algorithms, a lot of work over the last few years has gone into building scalable infrastructure, storage, and compression software to support the smooth running of these relatively simple algorithms. We've described the cutting edge for each of these categories earlier in this chapter. In this section, we will describe some of the new frontiers that search is exploring today as well as some areas we believe search will go towards.

Real Time Data and Search

The advent of services such as Twitter and Facebook in the last few years has made it extremely easy for individuals around the world to create information in the form of microposts. For ease of understanding, we will focus on Twitter data in this section. In 140 characters, users can describe where they are, publicize a link to an article they like, or share a fleeting thought. From a search engine perspective, this information is extremely valuable, but as different projects have shown over the last few years, this information has extra importance if it is mined and presented to search engine users in *real time*.

In order to build a real time system, the first prerequisite is access to the raw micropost data. Luckily for the community, Twitter has been extremely good about providing this at affordable prices to anyone who has requested it. Then comes the hard part, dealing with this firehose of data. At the time of this writing, there are ~90M tweets being generated daily, i.e. 1040 tweets per second [Rao, 2010]. At 140 bytes each, this corresponds to 12.6GB of tweet-data created in a day. After having dealt with petabytes of data, dealing with corpus of this size is trivial for any search engine today. The question to ponder on is what can we do with each tweet. Let's consider a few ideas:

1. Create a Social Graph: One of the beauties of data creation on services like Twitter is the fact that each user creates a graph of who they are interested in (who they follow) as well as the topics they are interested in (what they tweet about as well as topics in the tweets of the users they follow). The number of followers, too, is a good measure of how well respected that user is. Based on this knowledge, it is possible to create a graph of users as well as the thought leaders for different topic areas. We will refer to a user's influence on his her followers as UserRank and a user's influence on a given topic as UserTopicRank. Startups such as Topsy and Klout have built systems that expose this data.
2. Extract and index the links: Just like a traditional search engine, this would involve parsing each tweet, extracting a URL if present, crawling it and indexing it. The secondary inputs for the indexing stage are similar to those needed for webpage indexing. Instead of having anchor text, we have tweet-text. Instead of PageRank, we have UserRank. Additionally, we can use domain specific data that we have gathered from our web indexing to determine the quality of domains. All of the compression and storage algorithms will work without modification. Google, Microsoft, Cuil, OneRiot and Topsy are some of the companies that have worked in this direction.
3. Real-Time Related Topics: Related topics help users discover information about current topics better than traditional suggestions. For instance, on the day of the 2008 presidential debate in Kodak Theater, showing a related topic of 'Kodak Theater' for the query 'Barack Obama' would be much more meaningful than 'Michelle Obama'. However, a query of 'Byzantine Empire' would be better served with traditional related topics unless there was a new discovery about it.

There has been a lot of work on topic clustering [Kanungo, 2002]. Real-time data provides a very different use for these algorithms. By combining co-occurrence and information gain with a time decay factor, it is possible to analyze tweets and derive the related topics in real time are.

4. Sentiment Analysis: There are many teams such as Scout Labs and The Financial Times' Newssift team working on using NLP (Natural Language Processing) techniques to extract sentiment from tweets and other real time sources [Wright, 2009]. Whether this adds any value to marketing campaigns or feedback about products and services is yet to be seen.

Social Search and Personalized Web Search

The amount of interaction between users on the World Wide Web has increased exponentially in the last decade. While much of the interaction is still private (on email, chat, etc.), there has recently been a surge in public communications (via Twitter, Facebook, etc). The services providing such public communication platforms are also enabling third party applications to keep track of these social interactions, through the use of authentication APIs. Only a few companies, so far, have tried to improve user search experience through the use of social networking and related data. We discuss couple of such efforts below.

Over the last few years, Facebook has become the leader in social networking with over 500M users [Zuckerberg, 2010]. Facebook users post a wealth of information on the network that can be used to define their online personality. Through static information such as book and movie interests, and dynamic information such as user locations (Facebook Places), status updates and wall posts, a system can learn user preferences. Another feature of significant value is the social circle of a Facebook user, e.g. posts of a user's friends, and of the friends' friends. From a search engine's perspective, learning a user's social interactions can greatly help in personalizing the results for him or her.

Facebook has done two things that are impacting the world of search. First, in September 2009, they opened up the data to any third party service as long as their user authenticate themselves using Facebook Connect [Zuckerberg, 2008]. Second, as of September 2010, Facebook has started returning web search results based on the recommendations of those friends who are within two degrees of the user. The full description of this system can be found in their recently granted patent [Lunt, 2004]. In light of these recent developments, little has been done with this newly available data.

In late 2009, Cuil launched a product called Facebook Results [Talbot, 2009], whereby they indexed an authenticated user's, as well as his or her friends', wall posts, comments and interests. Noting that the average user only has 300-400 friends with similar preferences and outlooks on the world, one of the first discoveries that Cuil made was the fact that this data was extremely sparse. This implied that there were very few queries for which they could find useful social results. They overcame this limitation by extracting related query terms, which allowed for additional social results. A query for "Lady Gaga",for instance, would return friends' statuses that mentioned "Lady Gaga" but would also return any posts that mentioned songs such as "Poker Face" or related artists such as "Rihanna". Thus, using related data greatly enriched the user experience.

While social data is still relatively new to the web, there have been a few movements to use this data in ways that create value for web search. Klout, a San Francisco based startup, measures the influence of a user on his or her circle of friends and determines their 'klout score' (UserRank), as well as the topics they are most influential on (UserTopicRank) [Rao, 2010]. In the future, web search engines can use such a signal to determine authority of social data. In October 2010, Bing and Facebook announced the Bing Social Layer [Nadella, 2010] offering the ability to search for people on Facebook and to see related links that a user's friends had liked within Bing's search results.

CONCLUSION

This chapter describes in detail the key indexing technologies behind today's web-scale search engines. We first explained the concept of an inverted index and how it is used to organize all the web's information. Then we highlighted the key challenges in optimizing query processing time so that results are retrieved as fast as possible. This was followed by a discussion on using phrases over terms for better understanding of user intent in a query, along with its drawbacks for an indexing system. Harnessing together the power of multitudes of machines has been the key to success for today's search engines. Our key focus in this chapter has been to provide a better understanding of how these resources are utilized. We started with discussing the design tradeoffs for distributing data across a cluster of machines, specifically the cost of data transfers and index management. Next, we evaluated the different storage options to hold an index of web scale, specifically highlighting the impact of compression in dramatically shrinking index size and its effect on index updates. We also covered strategies that reorder an index for faster retrieval. This was followed by an overview on the infrastructure needed to support the growth of web search engines to modern scales. Finally, we closed the chapter with potential future directions for search engines, particularly in the real-time and social context. Recent efforts on these new data sources enrich the user's web search experience.

REFERENCES

Alpert, J., and Hajaj, N. (2008). *We knew the web was big…,* Google Blog, Retrieved on 10/13/2010 from http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html

Anh, V., and Moffat, A. (2005). *Inverted index compression using word-aligned binary codes.* IR 8(1):151–166.

Arrington, M. (2008). *Cuil Exits Stealth Mode With a Massive Search Engine.* Retrieved 10/13/2010 from http://techcrunch.com/2008/07/27/cuill-launches-a-massive-search-engine/

Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Lee, E., Morgan, N., Necula, G., Patterson, D., Sen, K., Wawrzynek, J., Wessel, D. & Yelick, K. (2008). *The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View*. Technical Report, UC Berkeley

Brin, S., Page, L. (1998). *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. Proceedings of the 7th international conference on World Wide Web (WWW). Brisbane, Australia. pp. 107–117. 1998

Chiang, T. , Chang, J., Lin, M. and Su, K. (1992). *Statistical models for word segmentation and unknown resolution.* In Proceedings of ROCLING-92. Pp. 121-146.

Computer School (2010). *Do you know how massive Google Is?* Retrieved on 10/13/2010 from http://www.99cblog.com/4739/do-you-know-how-massive-is-google-size-infographic

Cutts, M. (2006-02-02). *Confirming a penalty*. Retrieved on 10/13/2010 from http://www.mattcutts.com/blog/confirming-a-penalty/

Dean, J. and Ghemawat, S. (2004). *MapReduce: Simplified Data Processing on Large Clusters*. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004. Retrieved on 10/13/2010 from http://labs.google.com/papers/mapreduce.html

Elias, P. (1975). *Universal code word sets and representations of the integers.* IEEE Transactions on Information Theory 21(2):194–203.

Funt, P. (1998). *Extracting key terms from Chinese and Japanese texts.* Int. J. Comput. Process. Oriental Lang. Special Issue on Information Retrieval on Oriental Languages, 99-121

Ghemawat, S., Gobioff, H. and Leung, S. (2003). *The Google File System*. 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003. Retrieved on 10/13/2010 from http://labs.google.com/papers/gfs.html

GlusterFS (2010). *Gluster File System*. Retrieved 10/13/2010 from http://www.gluster.org

Golomb, S. W. (1966). *Run-length encodings.* IEEE Trans. Inform. Theory IT–12, 3 (July), 399–401.

Kanungo, T., Mount, D., Netanyahu, N., Piatko, C., Silverman, R., Wu, A. (2002). *An efficient k-means clustering algorithm: Analysis and implementation.* IEEE Trans. Pattern Analysis and Machine Intelligence, 24, pp. 881-892

Kapur, S., Parikh, J. and Joshi, D (2004) *Systems and methods for search processing using superunits*. US Patent Application 20050080795: Published April 14, 2005. Filed March 9 2004

Kesmodel, D. (2005-09-22). *Sites Get Dropped by Search Engines After Trying to 'Optimize' Rankings*. Wall Street Journal, Retrieved on 10/13/2010 from http://online.wsj.com/article/SB112714166978744925.html

Kleinberg, J. (1999). *Authoritative sources in a hyperlinked environment*. Journal of the ACM 46 (5): 604–632. 1999

Lunt, C., Galbreath, N., and Winner, J (2004). *Ranking search results based on the frequency of clicks on the search results by members of a social network who are within a predetermined degree of separation*. US Patent Application 10/967,609: Published August 31, 2010. Filed October 18, 2004

Manning, C., Raghavan, P., and Schütze, H. (2008) *Introduction to Information Retrieval*. Cambridge University Press.

Moffat, A., and Zobel, J. (1996) *Self-indexing inverted files for fast text retrieval.* TOIS 14(4):349–379.

MooseFS (2010). *Moose File System*. Retrieved 10/13/2010 from http://www.moosefs.org

Motzkin, D. (1994). *On high performance of updates within an efficient document retrieval system*. Inform. Proc. Manag. 30, 1, 93–118

Nadella, S. (2010-10-13). *New Signals In Search: The Bing Social Layer*. Retrieved on 10/13/2010 from http://www.bing.com/community/blogs/search/archive/2010/10/13/new-signals-in-search-the-bing-social-layer.aspx

Patterson, A.L. (2005). *We wanted something special for our birthday…* Retrieved 10/13/2010 from http://googleblog.blogspot.com/2005/09/we-wanted-something-special-for-our.html

Patterson, A,L. (2004), *Phrase-based searching in an information retrieval system*. US Patent Application 20060031195: Published February 9, 2006. Filed July 26, 2004

Porter, M.F. (1980) *An algorithm for suffix stripping*. Program, Vol. 14, No. 3. pp. 130-137

Pugh, W. (1990). *Skip lists: A probabilistic alternative to balanced trees.* CACM 33(6):668–676.

Rao, L. (2010-09-14). *Twitter Seeing 90 Million Tweets Per Day, 25 Percent Contain Links.* Retrieved on 10/13/2010 from http://techcrunch.com/2010/09/14/twitter-seeing-90-million-tweets-per-day

Rao, L. (2010-10-13). *Facebook Now Has Klout*. Retrieved on 10/13/2010 from http://techcrunch.com/2010/10/13/facebook-now-has-klout/

Rice, R. F. (1979). *Some practical universal noiseless coding techniques*. Tech. Rep. 79-22, Jet Propulsion Laboratory, Pasadena, CA.

Salton, G. (1962). *The use of citations as an aid to automatic content analysis*. Tech. Rep. ISR-2, Section III, Harvard Computation Laboratory, Cambridge, MA.

Salton, G., Wong, A., And Wang, C. S (1975). *A vector space model for automatic indexing*. Comm. ACM 18, 11 (Nov.) 613–620.

Scholer, F., Williams, H., Yiannis, J., and Zobel, J. (2002). *Compression of inverted indexes for fast query evaluation.* In Proc. SIGIR, pp. 222–229. ACM Press. 2002.

Silberschatz, G. (1994). *Operating System concepts*, Chapter 17 *Distributed file systems*. Addison-Wesley Publishing Company.

Singhal, A., Buckley, C., and Mitra, M. (1996) *Pivoted document length normalization.* In Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. Zurich, Switzerland, 21–29, 1996

Smith, A.J. (1985). *Disk cache—miss ratio analysis and design considerations.* ACM Transactions on Computer Systems (TOCS), v.3 n.3, pp. 161-203

Talbot, D. (2009-07-22), *Cuil Tries to Rise Again*. Retrieved on 10/13/2010 from http://www.technologyreview.com/web/23038/

Trotman, A. (2003). *Compressing inverted files.* IR6(1):5–19.
Wardrip-Fruin, Noah and Montfort, Nick (2003) (eds). *The New Media Reader. Section 54. The MIT Press*.

Witten, A.H., Moffat, A., and Bell, T.C. (1999). *Managing Gigabytes: Compressing and Indexing Documents and Images*, 2nd edition. Morgan Kaufmann.

Wright, A. (2009-08-23). *Mining the Web for Feelings, Not Facts.* The New York Times. Retrieved on 10/13/2010 from http://www.nytimes.com/2009/08/24/technology/internet/24emotion.html

Yahoo (2010). *The History of Yahoo! How It All Started…* Retrieved 10/13/2010 from http://docs.yahoo.com/info/misc/history.html

Zhang, J., Long, X., and Suel, T. (2007). *Performance of compressed inverted list caching in search engines.* In Proc. CIKM. 2007.

Zobel, J., and Moffat, A. (2006). *Inverted files for text search engines.* ACM Computing Surveys 38(2).

Zuckerberg, M. (2008-12-04). *Facebook Across the Web.* The Facebook Blog. Retrieved on 10/13/2010 from http://blog.facebook.com/blog.php?post=41735647130

Zuckerberg, M. (2010-07-21). *500 Million Stories*. The Facebook Blog. Retrieved on 10/13/2010 from http://blog.facebook.com/blog.php?post=409753352130

KEY TERMS & DEFINITIONS

Anchor Text: Visible, clickable text in a hyperlink.

Corpus: A collection of documents that are used for indexing.

Dictionary: A collection of terms used in an index.

Delta Encoding: Encoding technique that stores differences in values in a sorted array rather than full values.

Gamma Encoding: Technique used to encode positive integers when the upper bound is unknown.

Intersection: The operation of finding the overlapping elements of two sets. In the context of web search, posting lists are intersected for multi-term queries.

Inverted Index: A collection of posting lists.

Off Page Signals: Features extracted for a term on a given webpage from the contents of other webpages. Example: Presence of a term in anchor text.

On Page Signals: Features extracted for a term on a given webpage from the contents of that page itself. Example: Presence of term in Title.

Posting List: A list of dentifiers for documents that contain a given term.

Skip List: An auxiliary data structure to posting lists that enables skipping parts of it during intersection.

Term-Incidence Matrix: A boolean matrix indicating whether or not a given term appears in a given document.

Variable Byte Encoding: An encoding technique that uses a variable number of bytes encode an integer.