Knut Magne Risvik

Scaling Internet Search Engines : Methods and Analysis

A Dissertation submitted to the department of Computer and Information Science at the Norwegian University of Science and Technology for partial fulfillment of the requirements for the degree of Dr. Philos.

Trondheim, April 29, 2004

ISBN 82-471-6318-7 (trykt utgave)
ISBN 82-471-6317-9 (elektronisk utgave)

Abstract

This thesis focuses on methods and analysis for building scalable Internet Search Engines. In this work, we have developed a search kernel, an architecture framework and applications that are being used in industrial and commercial products. Furthermore, we present both analysis and design of key elements.

Essential to building a large-scale search engine is to understand the dynamics of the content in which we are searching. For the challenging case of searching the web, there are multiple dimensions of dynamics that should ideally be handled. In this thesis we start by examining some of these dimensions and the implications they have on search engine design.

When designing a search engine kernel, the focus has been on selection of algorithms and datastructures in the general case. Also, and even more important, we design worst-case characteristics into the search kernel that are very decisive from a scaling standpoint. A performance model to analyze the behavior of the kernel is also developed.

The designed search engine kernel was realized as a predecessor of the current FAST Search kernel (the FMS Kernel), and practical experiments and benchmarking demonstrate the correctness of the assumptions from the design of the kernel.

Then a framework for scaling shared-nothing systems based upon nodes working on separate portions of the data is introduced. The design of the framework is based on the general principles of replication and distribution. A performance model and an algorithm for cluster design are provided. This is in turn applied to construct a large-scale web search engine and benchmarking of clusters indicate that the assumptions and models for the distributed architecture hold.

The scaling aspect of search engine is further studied in the context of the application itself. Query locality is explored and used to create an architecture that is a generalized type of caching (through partial replication) using the application behavior and a configurable correctness trade-off to design super-linear scalable search engines.

Finally, a discussion of how linguistics are being used in web search engines is provided, focusing on the constraints that apply to ensure the desired scalability.

Contents

List of Figures

Acknowledgments

The development of the FAST Web Search system and the underlying technology has been a true team event. Being able to take part in the founding of this company and its technology has been a remarkable experience.

First and foremost, I would like to thank professor Arne Halaas for his passion,initiative and efforts to make both FAST and this thesis happen.

During the starting days at FAST, the teamwork with Tor Egge and Børge Svingen was a real source of inspiration, and their efforts have been essential to make this technology happen. I would also like to thank the entire FAST team in Trondheim for their commitment and support the last years and to making our technology a success.

During the short time we were part of Overture, I was very much inspired by working with the excellent teams both in Palo Alto and in Pasadena. I would especially like to thank Jan Pedersen and John Ellis, whose experience was inspiring and exciting.

Hugo Gunnarsen, Jeff Harlan and the entire FAST and Overture team in Sacramento have been extremely helpful for discussions, testing and benchmarking of both experimental and production systems.

I also appreciate the support from my colleagues in Yahoo (Qi Lu and Phu Hoang), and for giving me the necessary time to finish this work.

I thank Tomasz Mikolajewski for all input on the Linguistics chapter.

My mother always deserves a lot of thanks, for her encouragement and for believing in me.

Finally, thanks to my wife and daughter for their love and support – making anything worthwhile.

1 Organization

This thesis contains 3 published papers ([97], [93] and [44]), and two technical notes ([94] and [95]). [95] is further submitted for publication in IEEE Transactions on Knowledge and Data Engineering. Each of the papers are self-contained, discussing certain aspects around technology for web search engines, focusing on scaling and performance. Furthermore, three patents have been derived from the work found in this thesis ([91], [96] [92]).

The main section of this thesis is a thorough discussion of web search engines, and the techniques required to scale every component of such an engine, referring to the individual papers.

1.1 Readers Guide

Each of the papers included in the thesis are self-contained contributions, and could be read individually. The goal of the main section of this thesis is to connect the papers in the context of designing and building a scalable Internet search engine. Thus, the papers should be read before the main section.

2 Introduction

The area of search engines has had a tremendous growth and evolution over the last very few years with the increasing influence of the World Wide Web[11]. Search engines have grown from simple instantiations of early Information Retrieval models into multi-billion dollarbusinesses taking on enormous amount of traffic and becoming the single most important sources for information access on the web.

2.1 Models for Information Retrieval

Information Retrieval (IR) is the process of identifying and retrieving relevant documents based on a user's query. An IR system consists of three basic elements: a document representation, a query representation, and a measure of similarity between queries and documents. The document representation provides a formal description of the information contained in the documents; the query representation provides a formal description of user's information need; and the similarity measure defines the rules and procedures for matching the query and relevant documents.

These three elements collectively define a retrieval model. The most common models include the Boolean Model [111], the vector space model [99], the probabilistic model [111], and the inference network model [109]. We briefly outline the Boolean model and the vector space model below. The interested reader is encouraged to look into [111] and [109] for detailed studies of the models. A more detailed introduction to all models can be found in [8].

2.1.1 The Boolean Model

In Boolean retrieval [111], a document is represented as a set of terms $d_j = t1, \ldots, t_k$, where each t_i is a term that appears in document d_j. A query is represented as a Boolean expression of terms using the standard Boolean operators: *and*, *or* and *not*. A document matches the query if the set of terms associated with the document satisfies the Boolean expression representing the query. The result of the query is the set of matching documents.

2.1.2 The Vector Space Model

The vector space model [99] enhances the document representation of the Boolean model by assigning a weight to each term that appears in a document. A document is then represented as a vector of term weights. The

number of dimensions in the vector space is equal to the number of terms used in the overall document collection. The weight of a term in a document is calculated using a function of the form $tf \cdot idf$, where tf (term frequency weight) is a function of the number of occurrences of the term within the document and idf (inverse document frequency weight) is an inverse function of the total number of documents that contain the term.

A query in the vector space model is treated as if it were just another document allowing the same vector representation to be used for the queries as for documents. This representation naturally leads to the use of the vector inner product as the measure of similarity between the query and a document. This measure is typically normalized for vector length, such that the similarity is equal to the cosine of the angle between the two vectors. After all of the documents in the collection has been compared to the query, the system sorts the documents by decreasing similarity measure and returns a ranked listing of documents as the result of the query.

2.1.3 Hybrid solutions

Search engines typically use a hybrid mix of a boolean model and a vector space model. The boolean model is used for optional (or required) query syntax, while the rationale behind the vector space model is used for relevance calculations.

2.2 IR and Search Engines - a brief history of time

Information Retrieval by the means of computers has been a known concept since the beginning of the 1960's. This makes the research community centered around the problems of IR an experienced and advanced one.

The historical timeline of IR has been well documented several places, like [8], [111], and [99].

The applications of IR was restricted to specialized information systems where the setting was well controlled or at least controllable, i.e.:

- Content is homogeneous and known for the application. Coverage is finite.
- Users are of a known and limited numbers, receiving training to efficiently use the IR system.
- Content update rate is in many cases a known parameter of the application.

2.2.1 And then there was the Web

The introduction of the World Wide Web[11] created a totally different world of needs for IR. As the web started to grow, we had :

- Heterogeneous content, and an infinite coverage problem.
- Huge numbers of users, untrained, queries highly indeterminate.
- Update rate of content unknown.

All of this imposed a huge challenge for search and IR technology. Search engines grew from small but groundbreaking projects like WebCrawler [86], Google [21] and AltaVista[90] into a multi-billion dollar industry, and the second most popular application on the Internet (after email).

Along the path from the smaller projects, many challenges have emerged, some of them even partially solved. Challenges representing the difference between and standard IR system and a search engine for the web. A lot of work has been done on the task of combing Hypertext with IR. In [30] a model for retrieval taking the hypertext structure and links into account was introduced. Kleinberg [57] and also Page[85] abandoned much of the old retrieval model thinking, sorting retrieved documents by network analysis models.

2.3 The Role of a Search Engine

The role of search engines has clearly matured from being research experiments into being very important business engines for marketing and knowledge discovery. Search engines have found their place in every persons daily life on the Web, and is still growing into our desktop and workspaces.

In [76] am interesting perspective is taken, trying to evaluate the usefulness of a search engine response to a query. The results clearly indicate that there still is a huge potential for improving the overall quality of search engines. We certainly still are in the infancy of the Search Engine development.

2.4 The FAST Web Search Engine

The FAST Web Search Engine system is a large scale search engine offering OEM services to multiple customers. Currently, the engine holds more than 3 billion web pages in the index, with updates every two weeks. The system is running on more than 1000 nodes, and is capable of handling more than

800 queries per second. The crawlers have a URL list of more than 5 billion entries, and are currently touching more than 150 million pages every day.

Fast Search & Transfer (FAST) (http://www.fast.no/) is a Norwegian company that has been operating in the web search and corporate search segments since 1997. The web search division is probably most known for its search engine AllTheWeb (http://www.alltheweb.com/), and for supplying search results to portals in most parts of the world. After this thesis was drafted, the web search product was sold to the American company Overture. Overture has now been acquired by Yahoo, and the search technology from FAST is a key part of the Yahoo search platform.

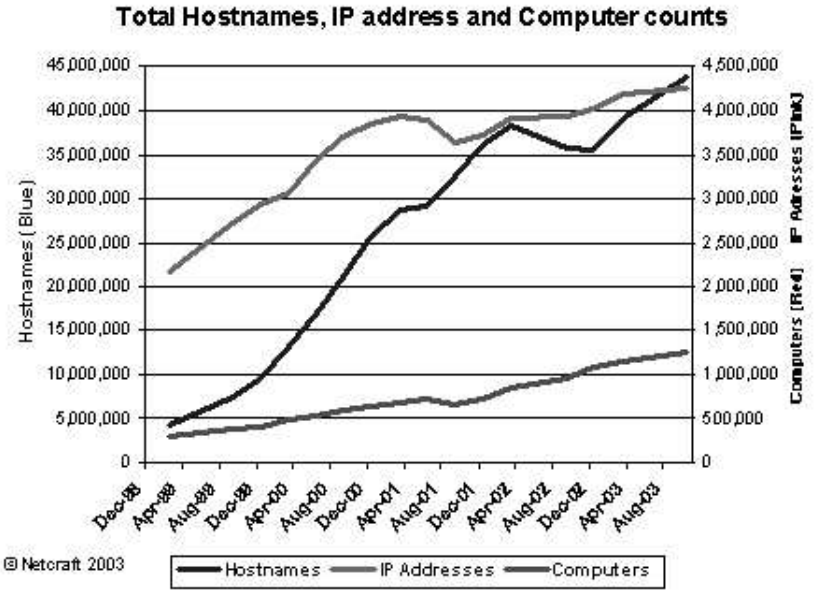


Figure 1. Hostnames, IP addresses and Computer count development

3 The Web and its Scaling Dynamics

To be able to build a useful search engine for the web, it is crucial to understand the dynamic characteristics of the web. In [97] we shed light onto some of the scaling issues for search engines on the web. Also in [10] there is a very detailed discussion around modeling of the dynamics of the web.

In this chapter, we touch upon some of the challenges and aspects of scaling that are mention in [97] and their recent development.

3.1 Growing in Size

The Web is reported to have had an exponential growth in every published paper about the topic. Both ISC [51] and Netcraft [83] perform domain and host surveys Figure 2, Figure 1, confirming the exponential growth. Giles and Lawrence introduced an overlap-based method for estimating the size of the web that seem to indicate the same in [63] and [64].

In [87] the concept of the Deep Web is introduced, suggesting that the

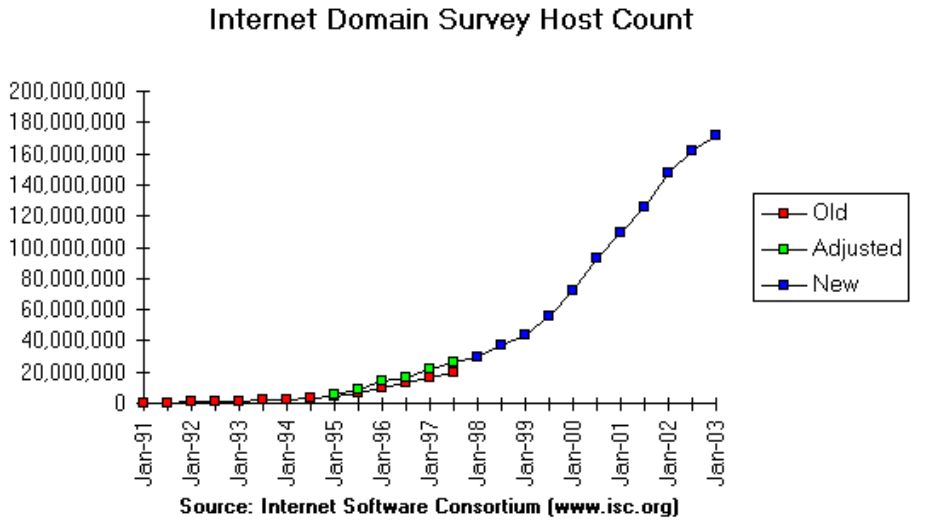


Figure 2. Domain name surveys

web is decades of pages larger than what is ever touched by search engines. The content referred to as the Deep Web is the content found behind e-commerce sites, collaborative filtering systems and other database sources that are not crawlable, but only accessible through a possibly personalized interface.

3.2 The Freshness

A dimension of the web that is very interesting for search engines is the freshness and update dynamics.

3.3 Growing in Content Diversity

This dimension of scaling is illustrated by the presence of other formats on the web, and the growing use of the Web for B2B application date. The Semantic Web [12] will also be a new dimension of the web.

4 Key Components of a Web Search Engine

A Web Search engine is a complex software and hardware system consisting of several components. In the papers in this thesis we have used a reference model of a web search engine to illustrate key components and the flow between these.

Definition 1 Crawler. *A crawler is a module aggregating documents from the World Wide Web in order to make them searchable. Several heuristics and algorithms exist for crawling, most of them based upon following links in hypertext documents.*

Definition 2 Indexer. *A module that takes a collection of documents or data and builds a searchable index from them. Common methods are inverted files, signature files, suffix structures and hybrids of these. In this thesis we will use the FMS Search kernel indexer system.*

Definition 3 Searcher. *The searcher is working on the output files from the indexer. The searcher accepts user queries from the dispatcher (defined below), executes a query over its part of the index, and returns sorted search results to the dispatcher with document ID and the relevance score.*

Definition 4 Dispatcher. *The dispatcher receives the query from the user, compiles a list of searchers to execute the query, sends the query to the searchers and receives a sorted list of results back from each searcher. For each result it receives a unique document ID, and the relevance score. The hits from the searchers are then merged to produce the list of results with the highest relevance scores for presentation to the user.*

In this thesis we will describe a search kernel along with its index structure [94] and a framework for scaling a search engine [95]. Together the modules cover the *Indexer*, *Searcher* and *Dispatcher* from the reference model above. In [97] the crawler part of the FAST Web Search is also briefly described.

4.1 Aggregation of Data – Crawling the Web

Crawling is still the dominant way of aggregating content for the main web search engines. Crawling the web imposes a large set of serious challenges to cope with the dynamics of web as outlined earlier in this thesis, and also covered in [97]. All major search engines use a crawler to gather documents

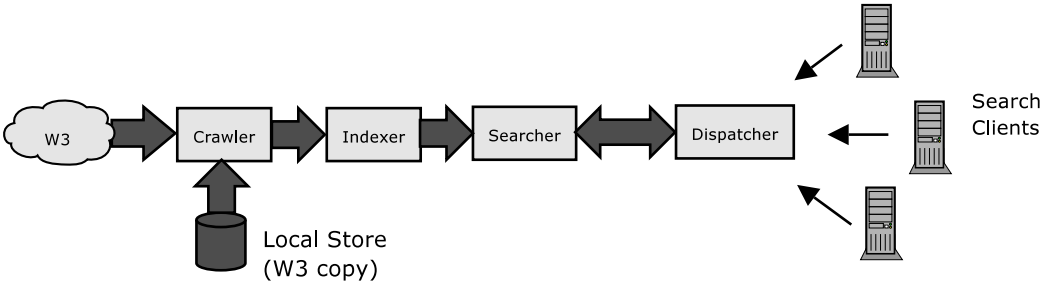


Figure 3. Search Engine reference model

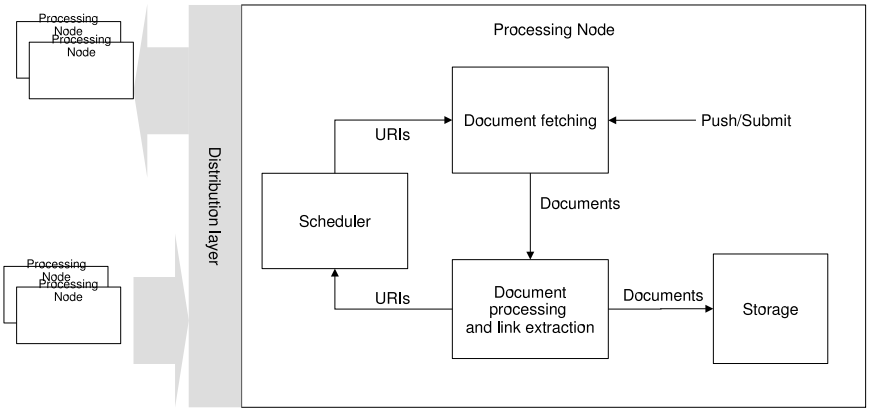


Figure 4. A Model for Crawler Systems

for indexing. The design of the crawlers for the commercial engines is not revealed with a lot of details, but some of them are discussed in [21] and [48].

A crawler system usually has multiple subcomponents, briefly touched upon below, and illustrated in Figure 4.

4.1.1 Scheduling

Scheduling what documents to download is a key algorithm of a crawling system. Performance in freshness, coverage and quality all depend on proper scheduling of content. Furthermore, one wishes to maximize bandwidth consumption of the crawler system. Ordering of pages are extensively discussed, among others in [27, 32, 106]. The dynamics of web pages is also a topic

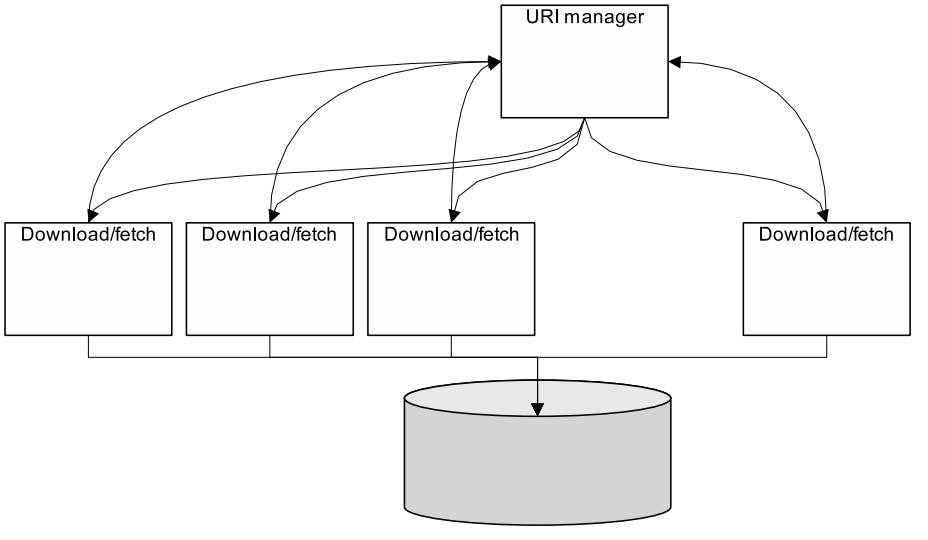


Figure 5. Crawler with centralized URI managers and storage

highly related to scheduling, and discussion are found in [19, 25].

4.1.2 Document Downloading and Storage Systems

Once the sequence of downloading has been determined, there is a subsystem to handle downloading of the content and usually storage for indexing purposes. [31, 41].

4.1.3 Scaling and Distribution Framework

Given the scaling and growth of the web, it has become imperative to parallelize or distribute crawling and aggregation of web pages for a large-scale search engine. Different crawlers seem to do that task in different ways. The early description of the Google crawlers [21] outlines that they have a centralized URL distributor that distributes download/crawl tasks to multiple slave nodes all working into a central storage. This is illustrated in Figure 5 However, the description of [41] shows a significantly more sophisticated storage part of the system.

The UbiCrawler introduced in[13] uses a fully distributed schema. By using *consistent hashing*[54] one avoids the issues of regular hashing, especially rehashing issues around adding buckets. The UbiCrawler schema has no centralized tasks, and its architecture is fault-tolerant by design. This is illustrated in Figure 6.

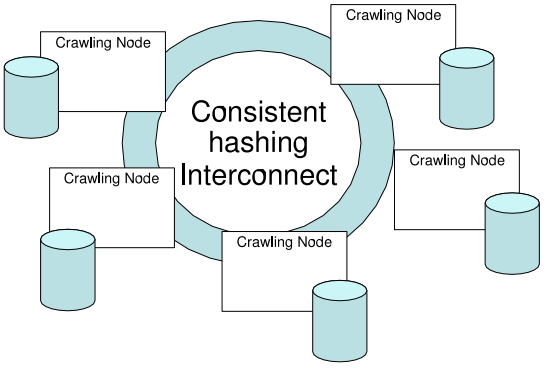


Figure 6. Fully distributed crawler

4.2 Building and Searching Indexes

Now, given a repository of web pages, the process of building a useful structure for searching is the next step (Figure 3). Many different indexing structures are being used for searching, however variants of inverted files [99, 113], suffix structures [71, 43] or signature files [37] are most common.

The amount of data to be indexed in a large-scale web search engine is of such a vast amount [63, 87], and has such dynamic characteristics [19, 97] that parallelization and distribution of the indexing and searching tasks is imperative to handling the systems.

For inverted files, there are two possible ways of distributing the index and the index building process:

- *Local Inverted Files.* Each node in the distributed systems holds an inverted index of a subset of the documents to be searched, so partitioning happens on the document identifiers.

- *Global Inverted Files.* Each node holds the inverted files for a subset of the terms in the dictionary.

Using local inverted files requires a synchronization step to ensure that dictionary statistics is normalized across the nodes, however on a system with a large set of documents, it seems to be the preferred distribution model. Also, suffix structures and signature files does not have the same distribution flexibility as the inverted files because the dictionary part is intertwined with the postings. FMS Search [94] uses a hybrid solution of indexing structures where the document based distribution is considered to

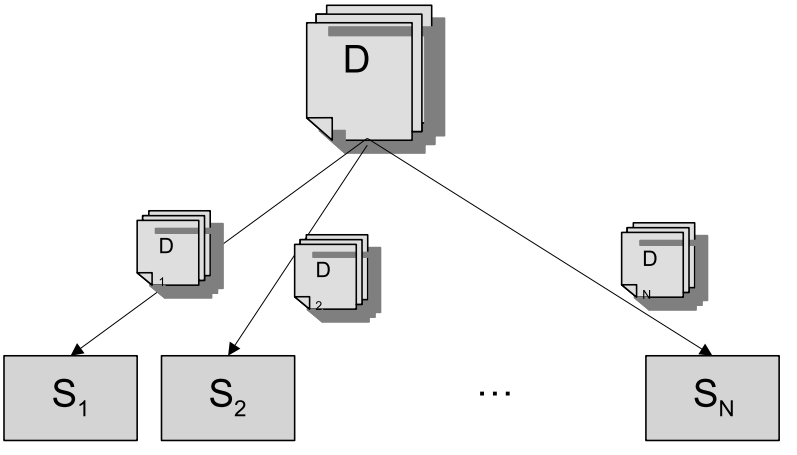


Figure 7. Document based distribution, assuming $\bigcup_{i=1}^{N} D_i = D$

be the best way, due to the intertwining of dictionary and postings. The document based distribution schema is briefly illustrated in Figure 7.

Index construction can either be incremental [23] or based on rebuilding the entire structure as most major search engines seems to be doing. In the rebuilding mode, one could either build new indexes on the active search nodes, or have offline search nodes for building the index, before swapping into online mode. These two options are briefly illustrated in Figure 8.

Creating an index usually has multiple steps involving parsing documents, sorting and merging dictionaries and postings. In [75] pipelining of these steps are discussed, and it is shown that this improves performance significantly.

4.3 Query Frontend

In the distributed cases outlined above, a query frontend to handle dispatching of queries and merging of results is required. However, in recent web search development there has also emerged technologies relying on preprocessing and analyzing queries as well as postprocessing the results coming back from the engine.

In [44] we outline some of the techniques that are often used on documents and queries, in some cases as a query preprocessing or a result postprocessing. In [98] we discuss how certain analysis could be used for dynamic relevance models.

A possible layer-model for the query frontend is shown in Figure 9, with

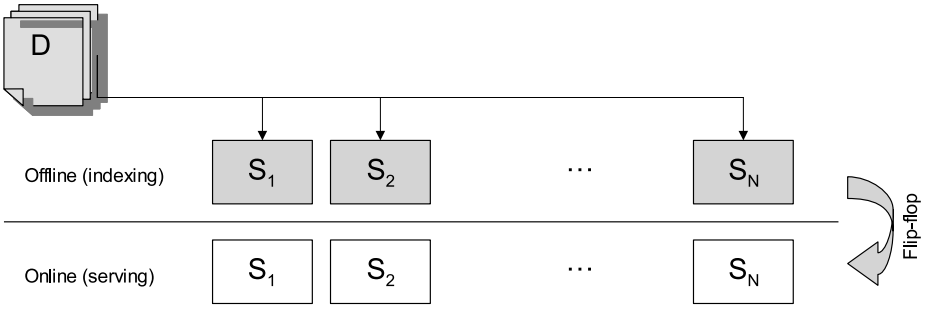


Figure 8. Having duplicated nodes for indexing and serving, with a flip-flop mode of switching

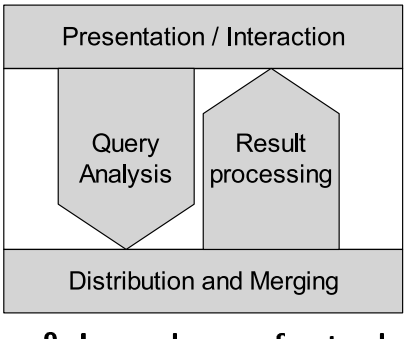


Figure 9. Layered query frontend model

3 layers:

- *Presentation/Interaction.* This layer holds the interaction and presentation parts of the engine. It can be from a simple HTML/XML template engine into a full-blown interactive search application.
- *Query and Result processing.* These steps usually are pipelines, in which queries are analyzed and possible rewritten or augmented with metainformation ($Q \rightarrow (Q'; M_Q)$), and where results are analyzed and also possible changed ($R \rightarrow R'$).
- *Distribution and merging.* This layer manages the distributed query serving nodes, ensuring all nodes receive the query and results are properly merged when creating the final search result.

5 The Quest for Scaling

Maintaining a web search engine requires studying the scaling properties of the web. The growth, the existence of the "deep web" [87], and the dynamics [19] are all challenging tasks to handle individually. A good search engine needs to take on all the challenges in a balanced manner [18].

This thesis focuses on scaling issues for the query serving part of a search engine system. This chapter will start by discussing design and performance characteristics of a search kernel, then move on to discuss a general architecture for scaling in size and query capacity. Furthermore, we discuss a new concept called tiering, and how that can improve the size and capacity scaling properties. Finally, we discuss linguistic technologies in a large-scale search engine, and what properties are required on those processes to work in a large-scale setting.

The FMS Search kernel [94] and the FAST DPA [95] in the setting of the FAST Web Search system is being used as the case throughout the papers encapsulated in this thesis.

5.1 Search Core Design and Performance

The design of commercial search kernels has not been public information at large, but in general it seems like most of the engines use inverted files or variants including suffix structures. The Google kernel is partially discussed in [21], and in this thesis (and [94]) we outline the basic structures and algorithms for the FMS Search kernel, a predecessor of the FAST Search system.

The search core is the foundation for building a scalable search engine system. In this section we will outline some basic requirements for a search core, and illustrate possible ways to meet those requirements using the FMS Search kernel as an example.

5.1.1 Ensuring Scalability of Search Kernel

Building a search kernel exposes a series of design option and possibilities. Everything from retrieval model to actual data structures and algorithms and load handling are critical for ensuring the wanted scalability. In this chapter we focus on three key challenges to ensure scalability:

- *Balancing I/O and processing.* Ensuring that we are not saturated in only processing or I/O is essential to ensure that the system is

operating in a sweet-spot area. The goal of the design is to make this area as wide as possible.

- *Memory versus Disk.* Disk and memory usage must be balanced in a way that is optimal with regards to cost and performance.

- *Worst-case handling.* Different queries can have various processing and I/O requirements. To avoid having queries interfere with other queries, there must be some worst-case handling of queries.

5.1.2 Balancing I/O and Processing

The balance between I/O and processing depends on choices of preprocessing and online processing, and also the selection of datastructures and algorithms.

Inverted files have been thoroughly studied in the literature, and their performance is well known both for querying and for building[116, 79, 113, 8].

The FMS search kernel deploys multiple structures, based on what query operations it should support (single term or phrase query). For single term queries (the mapping from a query term to a document, $q \rightarrow \{id_{doc}\}$, a simple inverted file with a separate dictionary is being used (the mapping from a query to a set of query terms, $Q \rightarrow \{q\}$. The inverted file does not store position occurrences, just frequency information for relevance ranking on each document-term pair.

A wide variety of compression schemes have been studied for inverted files ([113, 115]). The indexing structure as described in [94] requires 56 bits per posting in the basic implementation. Given the findings in [94] regarding dictionary growth being close to linear (but with a very low constant factor), the structure is relatively small (less than the corpus size). The processing required for a $q \rightarrow \{id_{doc}\}$ matching operation is very low, since it will be a single iteration over the inverted list for the given query term, q, with an aggregate relevance function to compute the relevance score for each entry, then followed by a final sort for the entire result set.

The phrase matching operation of a search engine is heavily used (through more advanced automatic query optimizations and relevance improvements) and the performance of this operation is critical. While the inverted lists for the single term matching operation is fairly small in size (since there is maximum one entry per document for each term), the use of inverted postings (with position information) for phrase matching could impose several serious issues with regards to performance:

- I/O load. The I/O load would be high for frequent terms (so-called stop words). It could also require to do block-wise operations on the inverted list, since it could possibly be too large to fit in memory.

- Processing load. The processing load for scanning very long posting lists (for each term) doing the merge with proximity operations could be heavy for certain cases where the individual terms in the phrase are frequent.

The FMS Search kernel aims to handle these issues by using a slightly different indexing structure for phrase matching ($q \rightarrow \{id_{doc}\}$). The datastructure is based on the ideas of suffix structures, more specifically the Suffix Array (discovered in parallel in [71] and [40]). Online construction of these structures were explored in [110] and [42].

Based on the suffix tree, a more space-economical structure was created, namely the sparse suffix tree [55].

The principle of the phrase matching structure of FMS Search is to have a sparse suffix array, only indexing the entries starting at word boundaries. (Since we only search for whole words in the query model). Furthermore, instead of using the backlink method introduced in [71], we store entire suffixes, with a cut-off at length 3. The relative position of the triple is also stored. Thus, one would divide a query into triples and query for the triples in parallel before intersecting to create the final resultset.

This method is a generalization of the nextword structures discussed in [112] and [9].

The method sacrifices relatively large storage requirements (about 200% of the raw data volume) for low I/O load and efficient processing. The performance model in [94] suggests that the query algorithm over the structure is sub-linear, and the experiments with queries including phrase queries seem to indicate the optimality of this model.

Furthermore, the bitvector mechanism, combined with the drilling system ensures an upper I/O bound.

5.1.3 Memory versus Disk

The balancing act of memory usage and memory size in concordance with disk I/O operations is also essential for scaling and performance of a search kernel.

In the FMS Search kernel, this balancing is assisted by three mechanism, namely caching, the use of *bitvectors* and drilling.

Web Search query logs inherit a clear locality feature ([93, 114, 100, 73]. Given the datastructures used in FMS Search, there is a very direct relationship between results and postings stored in the structure. Thus, the locality observed on the queries will map directly onto locality on the posting entries as well.

The FMS Search kernel [94] has both query caching as well as caching of the posting entries and the dictionary entries. The cache hit ratios found indicate the importance of this.

The *bitvector* structure enables us to do queries of stopwords without any ranking information with very limited memory usage, and also enabling bit-parallelism in boolean operations. Combined with the drilling mechanism, FMS Search ensures that memory balancing with disk usage can always be optimized. Bitvectors can be further optimized by using compression, as outlined in [77]. Also in [116] a discussion of bitvectors and bitslice signatures can be found, further optimizing the usage of bit-based structures.

5.1.4 Worst-case handling

Inverted files and the refined structures used in FMS Search has linear space requirements to the data volume being indexed. Still, one must be efficient on I/O, memory and CPU usage during query processing to reduce the impact of "hard" queries on the overall system performance.

FMS Search has several mechanisms for handling worst-case queries, and limiting their impact on system performance.

The *bitvectors* used for hit representation clearly limits the memory and processing requirements for queries or query terms with high frequency. However, the direct use of bitvectors would imply that no relevance ranking would be available for these queries or query terms.

In order to allow for a more flexible schema for impact reduction, the "drilling" system outlined in [94] allows us to define relative importance in subsections of documents, and to use ranking information only for subsections when the frequency of the query or the query term is considered too high. This allows for a more continuous reduction of ranking information than the binary scheme would be for bitvectors.

Furthermore, the FMS Search kernel employs a somewhat simplified $tf \cdot idf$ ranking function that is cheaper to compute. There are other ways of improving relevance ranking for worst case handling, also discussed in [113, 78, 80].

5.2 Scaling in Size and Capacity

A Web Search engine faces challenges that requires very good scaling capabilities in terms of size and capacity. In parallel computing, there are four typical models for computing (SISD, SIMD, MISD, MIMD). There has been a clear trend in parallel and cluster computing to move towards inexpensive and commodity based systems.

In IR, the distributed system approach has been a well-known one [70, 108, 89]. A large-scale search engine was also well-known for using a distributed scheme, namely the Inktomi engine [39] and [17].

5.2.1 Scaling Dimensions

For a web search engine, we see the importance of scaling in many dimensions. Here we focus on two dimensions:

1. *Data volume scaling.* Being able to handle a rapidly increasing amount of data to search is critical. Linear scaling in data volume is an absolute requirement.

2. *Query capacity scaling.* A publicly available search engine will often have to handle many million queries per day, and there is a strict requirement of interactivity.

Scaling in freshness is also a critical dimension, given the trends of the web [19]. However, this is considered to be outside the scope of this thesis.

5.2.2 Distribution Schemes

Distribution in IR is tied into the selected data structures of the search kernel. For inverted files, there have been two common ways of distribution, namely *local inverted files* and *global inverted files.* The first strategy is to distribute documents across the available processing nodes, making each node responsible for querying a subset of the documents. A merging framework on top of the nodes will be required to merge the hit lists.

The other strategy is to distribute the inverted files such that each node handles inverted lists for a subset of query terms. This also requires a merging framework on top, but with more complexity added to merge with proper combination of results. Performance studies [108] suggest that local inverted files in most cases have good performance and resource usage characteristics.

The datastructures used in FMS Search [94] are more complex than just inverted files, and the data volumes for a web scale search engine are

so large that one may assume that a uniform distribution of documents would be possible. Thus, using an approach where distribution is based on documents rather than terms is chosen.

5.2.3 FAST Web Search Scaling

In [95] a general framework for scaling in two dimensions, namely in size of data and in capacity of processing. Furthermore, a performance model is outlined and utilized to describe the FAST Web Search engine and to analyze its performance. Also, the FAST DPA includes models for fault-tolerance and ensuring desired levels of this.

The DPA scales in two dimensions using :

- *Replication* of processing capacity and data for increasing query capacity, and
- *distribution* of documents for handling data volume growth.

These are very basic methods for scaling [107].

The application of FAST DPA for Web Search illustrates the usefulness of the framework and of the provided design algorithms.Various configurations of nodes and clusters with the FMS Search kernel on each of the nodes was tested. As expected the scaling in data volume does not have a noticeable impact on the query capacity for the system. The latency in the system is a logarithmic function of the query complexity, but the growth of this is controllable through changing the fanout of the system. In our experiments the latency was well within the limits of any service level that would be required by an industrial application of a web search engine.

5.3 Caching and Tiering, Adaptive Scaling

[95] suggests an architecture that enables a search engine to scale linearly in either size or query capacity. However, the architecture is very general, and it applies to a large set of problems with shared-nothing characteristics (not only searching).

Caching of search engines results has been studied in multiple papers[65, 4, 73] clearly showing high locality in queries. Caching is an obvious way of improving scaling performance of search engine that benefits from high query locality. The locality is also supported by query log analysis like the ones found in [93] and [101]. In [107] a discussion of caching for IR system is also found.

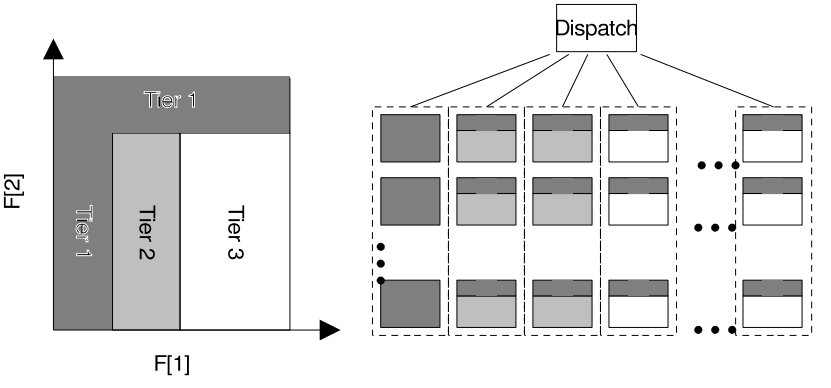


Figure 10. Mapping from a document space into tiers of search nodes

The use of partial replication for IR system was introduced in [68] and [67]. Partial replication means to replicate a smaller portion of the database and query that smaller replica first. The selection of the replica was proposed to be tied into a query log analysis, exploiting the locality. [69] analyzes the partial replication solution compared to a caching solutions and shows advantages of using partial replication.

In [93] we generalize the partial replication concept into a so-called *tiered architecture* where a collection of documents are grouped based on features along many dimensions (locality of query hits could be one). The mapping concept is illustrated in Figure 10 where documents are mapped from a feature vector with multiple dimensions to a set of search nodes (or possibly a sub-index on a set of search nodes). The sets of documents are disjoint, so $\bigcup_{i=1}^{N} T_i = D$ where T_i is defined as a tier or a subset of documents. The mapping from documents into tiers is done by a function, $\mathcal{F}$. Furthermore, a custom *fall-through algorithm, FTA* is used to determine how many tiers to search and when to fall through between them.

The tiering architecture can be configured in many ways by tuning the mapping function and the fallthrough algorithm. In the paper [92], we test three different alternatives based on three assumptions of relevance and locality:

1. Overall ranking is highly correlated into static ranking of document (PageRank, EigenRank etc).

2. Query locality, called *tier-locking* in the paper.

3. Term Context boosting is highly correlated into ranking. (Title and anchor hit boosting).

Studies in the paper clearly shows that the first assumption is incorrect, at least for the ranking we employ. The error numbers between the 1D-tier model and the reference model is very high.

By adding locality preferences, or tier-locking, the error rate drops significantly, which is what would be expected from published query and cache analysis papers [68, 4, 73, 65].

Furthermore, as suggested in [85] and [21] the anchor texts of links pointing a document have high importance when a match of the query is found there. The same assumption seems to be valid for title matches in Web Searches [46]. This is the foundation for the third assumption, we see from the 2D tiering solution that the error rate has dropped significantly and is approximating the reference system very well.

The performance improvements from the tiering architecture are superior to what you would expect from caching solutions. Furthermore the dynamics of the tiering architecture (the mapping and the fallthrough algorithm) enables one to tune the accuracy versus performance freely.

5.4 Linguistic Processes

Linguistics has been identified to be one of the central elements of the IR evolution [66, 103, 56]. One differentiates roughly between statistical (mostly language independent) [20] and grammatical (language sensitive) approaches to processing natural language. In the former branch, the popularly used $tf \cdot idf$ relevance ranking function is utilizing a "general vs. specific' linguistic relation of index terms. The latter branch of linguistic methods may be utilized in extracting 'grammatically specified' meaningful index terms.

More sophisticated linguistic techniques have entered the scene for information retrieval systems, and web search engines in particular. Taking into account the scale and scaling requirements of web search engines, there is a clear need to ensure that all applicable computational linguistics methods will have extreme scaling needs.

In [44] we describe the use of linguistic processing in the FAST Web Search engine, including language detection, lemmatization, phrase detection, text categorization and clustering.

There are currently two main applications for text categorization: language identification (i.e. categorizing the text as 'written in language X')

and detection of offensive content. These both methods are dictionary based i.e. each meaningful text sequence (word or phrase) is looked up in a dictionary for its vector of (category,weight) pairs. High level of efficiency in looking up a sequence is achieved by using finite state devices [81]. This enables categorization to happen online during either crawling or indexing.

Lemmatization is a part of the normalization of index terms. The approach outlined in [44] is again based on finite state automata for high enough performance to be part of the indexing process. Depending on how many documents a language is represented in, some more time consuming approaches may be used that improve the precision and recall e.g. decompounding for German and Scandinavian languages.

A very important linguistic technology used in the FAST Web Search engine is the phrasing capability. Based on considerably huge dictionaries of offline extracted phrases, we rewrite queries online into a multi-word entities (e.g. new york $\rightarrow$ 'new york'). As discussed, the quality of phrasing is coupled with the size of the phrase dictionary being used. The growing use of phrases or automatic (weighted) phrasing [98] clearly indicates the importance of efficient phrase matching structures in the search kernel. This was a clear design objective when building the FMS Search kernel [94], and also for the FAST Web Search system.

The technology for clustering of results described in the paper is an experimental service. Its quality depends again on the quality of extracted index terms with the techniques described above. From [95] the bandwidth requirements of the dispatching network system will be a bottleneck when sending large sets of data for each hit (e.g. the entire document, or possibly improving it by sending its most characteristic extracted index terms). The method can balance between precomputing feature vectors offline from the search process or compute the features on the fly from the document itself or from a summary of the document. Alternative document teasers that show the most relevant index terms for a hit document (table of concepts) may also be generated by using grammar based index term extraction. The quality of the clusters and the usability of clustered results are tightly coupled into how many documents of which we perform the clustering over and how good the considered index terms are. Typical ranges are $50 - 200$ documents and most of the index term grammars are developed (precomputed) for the most frequent languages.

6 Results and Future Work

This thesis is the results of many years of work by a group of people with a desire to build a true scalable search engine. This thesis wraps up the design and analysis of the key components we have built.

6.1 Contributions

1. We have discussed in a broad setting how characteristics of the dynamic web impose constraints and challenges for a large-scale search engine, touching on crawling, but focusing on indexing and searching.

2. The FMS Search kernel (and its sucessors) has proven in industry to be extremely scalable and to have excellent performance characteristics. This theses outlines the design process and choices to build the FMS Search kernel. Furthermore, a performance model is provided and experiments clearly suggests the correctness of that model. The FMS Search kernel has several mechanism that seem to be novel for these kind off applications, like hybrid suffix structures and the drilling mechanism to achieve the desired performance characteristics.

3. The FAST DPA framework is a general framework for shared-nothing parallel computing. Our contribution is to provide this along with a performance model and an algorithm to construct the clusters. The use of this model and algorithm on a large-scale web search engine shows the usefulness of this work, and the flexibility of the framework.

4. Even with linear scaling, we are approaching boundaries of technology and financial power in order to build search engine solutions capable of scaling with the web. The extreme locality found in query logs makes caching a critical technology to scale more efficiently. In this thesis we introduce tiering as a generalized version of partial replication. Tiering allows for the same performance effects as caching, but tiering also allows us to trade correctness for performance in a controlled manner. This makes sense, since the relevance ranking scores are very much approximations of relevance, and introduction of stochastic noise does not seem to destroy the user experience.

5. Linguistic technology is gaining importance for search engines. The ultimate goal of true natural language processing might still be some way ahead, but various linguistic processing has entered the search engine arena. In this thesis we discuss some of these processes for

a large-scale search engine, and present some constraints to ensure proper scaling.

6.2 Future Work

Search engines have become a high-target on the web radar screen both from a usefulness standpoint and from a financial standpoint. It is obvious that the area will be a crowded one for future research. The good news is that a huge set of research problems needs to be worked on further.

6.2.1 Search Engine Kernel Development

Development of search engine kernels will still be very important. A lot of search engine applications are built upon legacy-class search kernels. Introduction of incremental updates to a large index, much better support for fielded or structured searching and a more flexible ranking framework are some of the areas that could be brought together to realize a new search kernel.

6.2.2 Scalable Architectures and Tiering

The scalability of search engines is based on general scaling principles. The introduction of tiering takes application specific elements into account to create a super-linear scaling. An interesting study would be to evaluate tiering compared to caching, and also to study how much caching could improve performance on an already tiered system.

In order to study more dimensions of possible tiering, machine learning techniques should also be applied.

6.2.3 The Freshness Dimension

A dimension of the web dynamics that no search engine has taken fully into its operation is the freshness aspect. Most engines does snapshot updates of the index, and vast batch processing. A true incremental engine would be required. However this would require much more online processing, different technology for aggregation and a search engine kernel that allows for almost real-time updates of the index.

References

[1] AllTheWeb. http://www.alltheweb.com.

[2] AltaVista. http://www.altavista.com.

[3] Html engine. http://www.cs.colorado.edu/home/mcbryan/Home.html.

[4] Adali, S., Candan, K. S., Papakonstantinou, Y., and Subrahmanian, V. S. Query caching and optimization in distributed mediator systems. In *SIGMOD Conference* (1996), pp. 137–148.

[5] Ambroziak, J., and Woods, W. Natural language technology in precision content retrieval. In *Proceedings of the International Conference on Natural Language Processing and INdustrial Applications (NLP+IA'98)* (1998).

[6] Anderson, T. E., Culler, D. E., and Patterson, D. A. A case for NOW (Networks of Workstations). *IEEE Micro 15*, 1 (Feb. 1995), 54–64.

[7] Aron, M., Sanders, D., Druschel, P., and Zwaenepoel, W. Scalable content-aware request distribution in cluster-based network servers. In *Proceedings of the USENIX 20 Annual Technical Conference* (San Diego, CA, June 2000), pp. 323–33.

[8] Baeza-Yates, R., and Ribeiro-Neto, B. *Modern Information Retrieval*. Addison-Wesley-Longman, May 1999.

[9] Bahle, D., Williams, H. E., and Zobel, J. Efficient phrase querying with an auxiliary index. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval* (2002), ACM Press, pp. 215–221.

[10] Baldi, P., Frasconi, P., and Smyth, P. *Modeling the Internet and the Web: Probabilstic Methods and Algorithms*. Wiley, 2003.

[11] Berners-Lee, T., Cailliau, R., Luotonen, A., Nielsen, H. F., and Secret, A. The world-wide web. *Communications of the ACM 37*, 8 (1994), 76–82.

[12] Berners-Lee, T., Hendler, J., and Lassila, O. The semantic web. *Scientific American* (may 2001).

[13] Boldi, P., Codenotti, B., Santini, M., and Vigna, S. Ubicrawler: A scalable fully distributed web crawler, 2002.

[14] Bowman, C. M., Danzig, P. B., Hardy, D. R., Manber, U., and Schwartz, M. F. The Harvest information discovery and access system. *Computer Networks and ISDN Systems 28*, 1–2 (1995), 119–125.

[15] Bowman, C. M., Danzig, P. B., Manber, U., and Schwartz, M. F. Scalable Internet resource discovery: research problems and approaches. *Communications of the ACM 37*, 8 (1994), 98–107.

[16] Bray, T. Measuring the web. In *Proceedings of the Fifth International World Wide Web Conference (WWW5)* (1996).

[17] Brewer, E. A. Delivering high availability for inktomi search engines. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA* (1998), L. M. Haas and A. Tiwary, Eds., ACM Press, p. 538.

[18] Brewington, B., and Cybenko, G. Keeping up with the changing web. *IEEE Computer 33*, 5 (May 2000), 52–58.

[19] Brewington, B. E., and Cybenko, G. How dynamic is the Web? *Computer Networks (Amsterdam, Netherlands: 1999) 33*, 1–6 (2000), 257–276.

[20] Brill, E. Processing natural language without natural language processing. In *Proceedings of CICLin 2003, 4th International Conference on Computational Linguistics and Intelligent Text Processing* (Mexico City, Mexico, February 2003), Springer Verlag, Heidelberg, pp. 360–369.

[21] Brin, S., and Page, L. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th WWW Conference* (http://decweb.ethz.ch/WWW7/1921/com1921.htm, 1998).

[22] Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., and Wiener, J. Graph structure in the web. In *Proceedings of the Ninth International World Wide Web Conference (WWW9)* (2000).

[23] Brown, E., Callan, J., and Croft, W. Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB)* (Santiago, Chille, September 1994), pp. 192 – 202.

[24] Cho, J., and Garcia-Molina, H. Estimating frequency of change. Unpublished, 2000.

[25] Cho, J., and Garcia-Molina, H. The evolution of the web and implications for an incremental crawler. In *Proceedings of the Twenty-sixth International Conference on Very Large Databases* (2000).

[26] Cho, J., and Garcia-Molina, H. Syncronizing a database to improve freshness. In *Proceedings of the 2000 ACM International Conference on Management of Data (SIGMOD)* (2000).

[27] Cho, J., García-Molina, H., and Page, L. Efficient crawling through URL ordering. In *Proceedings of the Seventh International World Wide Web Conference (WWW7)* (1998).

[28] Cole, R., Mariani, J., Uszkoreit, H., Zaenen, A., and Zue, V. Survey of the state of the art in human language technology, 1995.

[29] Craswell, N., Hawking, D., and Griffiths, K. Which search engine is best at finding airline site home pages?

[30] Croft, W. B., and Turtle, H. A retrieval model incorporating hypertext links. In *Proceedings of the second annual ACM conference on Hypertext* (1989), ACM Press, pp. 213–224.

[31] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (Chateau Lake Louise, Banff, Canada, Oct. 2001).

[32] Diligenti, M., Coetzee, F., Lawrence, S., Giles, C. L., and Gori, M. Focused crawling using context graphs. In *26th International Conference on Very Large Databases, VLDB 2000* (Cairo, Egypt, 10–14 September 2000), pp. 527–534.

[33] Dougherty, E. R. *Probability and statistics for the engineering, computing, and physical sciences*. Prentice Hall, 1990.

[34] Economist Magazine. E-Entertainment survey, October 2000.

[35] EDWARDS, J., MCCURLEY, K. S., AND TOMLIN, J. A. An adaptive model for optimizing performance of an incremental web crawler. In *World Wide Web* (2001), pp. 106–113.

[36] ETESTING LABS. Fast Search and Transfer, Inc. web search engine evaluation. Tech. rep., 2001.

[37] FALOUTSOS, C., AND CHRISTODOULAKIS, S. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Office Information Systems 2*, 4 (October 1984), 267–288.

[38] FLYNN, M. J. Very high-speed computing systems. *Proceedings of the IEEE 54* (1966), 1901–1909.

[39] FOX, A., GRIBBLE, S. D., CHAWATHE, Y., BREWER, E. A., AND GAUTHIER, P. Cluster-based scalable network services. In *Symposium on Operating Systems Principles* (1997), pp. 78–91.

[40] G. GONNET, R. BAEZA-YATES, T. S. Lexiographical indicies for text: Inverted files vs. pat trees, 1991.

[41] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (2003), ACM Press, pp. 29–43.

[42] GIEGERICH, R., AND KURTZ, S. From ukkonen to mccreight and weiner: A unifying view of linear-time suffix tree construction. *Algorithmica 19*, 3 (1997), 331–353.

[43] GROSSI, R., AND VITTER, J. S. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). pp. 397–406.

[44] GULLA, J. A., AURAN, P. G., AND RISVIK, K. M. Linguistics in large-scale web search. In *Proceedings of the Natural Language Processing and Information Systems, 6th International Conference on Applications of Natural Language to Information Systems, NLDB 2002, Stockholm, Sweden, June 27-28, 2002, Revised Papers* (2002), B. Andersson, M. Bergholtz, and P. Johannesson, Eds., vol. 2553 of *Lecture Notes in Computer Science*, Springer.

[45] HAWKING, D., CRASWELL, N., BAILEY, P., AND GRIFFIHS, K. Measuring search engine quality. *Information Retrieval 4*, 1 (2001), 33–59.

[46] HAWKING, D., CRASWELL, N., THISTLEWAITE, P., AND HARMAN, D. Results and challenges in Web search evaluation. *Computer Networks (Amsterdam, Netherlands: 1999) 31*, 11–16 (1999), 1321–1330.

[47] HEAPS, H. S. *Information Retrieval: Computational and Theoretical Aspects*. Academic Press, 1978.

[48] HEYDON, A., AND NAJORK, M. Mercator: A scalable, extensible web crawler. *World Wide Web 2*, 4 (1999), 219–229.

[49] HIRAI, J., RAGHAVAN, S., GARCIA-MOLINA, H., AND PAEPCKE, A. WebBase: a repository of Web pages. *Computer Networks (Amsterdam, Netherlands: 1999) 33*, 1–6 (2000), 277–293.

[50] HUBERMAN, B. A., AND ADAMIC, L. A. Evolutionary dynamics of the World Wide Web. Tech. rep., Xerox Palo Alto Research Center, 1999.

[51] INTERNET SOFTWARE CONSORTIUM. Internet Domain Survey.

[52] JANSEN, B. J., SPINK, A., AND SARACEVIC, T. Real life, real users, and real needs: a study and analysis of user queries on the web. *Information Processing and Management 36*, 2 (2000), 207–227.

[53] JUDSON, T. W. *Abstract Algebra*. PWS Publishing Company, 1994.

[54] KARGER, D., LEHMAN, E., LEIGHTON, T., LEVINE, M., LEWIN, D., AND PANIGRAHY, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *ACM Symposium on Theory of Computing* (May 1997), pp. 654–663.

[55] KARKKAINEN, J., AND UKKONEN, E. Sparse suffix trees. In *Proceedings of COCOON, HongKong, 1996* (1996).

[56] KARLGREN, J. The basics of information retrieval: Statistics and linguistics.

[57] KLEINBERG, J. M. Authoritative sources in a hyperlinked environment. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (http://simon.cs.cornell.edu/home/kleinber/auth.ps, 1998).

[58] KOBAYASHI, M., AND TAKEDA, K. Information retrieval on the web. *ACM Computing Surveys 32*, 2 (2000), 144–173.

[59] KONTZER, T. In search of business data. *InformationWeek.com* (January 2002).

[60] KOSTER, M. A standard for robots exclusion.

[61] LANGER, S. Natural languages and the World Wide Web.

[62] LAWRENCE, S. Context in web search. *IEEE Data Engineering Bulletin 23*, 3 (2000), 25–32.

[63] LAWRENCE, S., AND GILES, C. L. Searching the World Wide Web. *Science 280*, 5360 (1998), 98–100.

[64] LAWRENCE, S., AND GILES, C. L. Accessibility of information on the web. *Nature 400*, 6740 (1999), 107–109.

[65] LEMPEL, R., AND MORAN, S. Predictive caching and prefetching of query results in search engines. In *Proceedings of the Twelfth International World Wide Web Conference (WWW2003)* (2003).

[66] LEWIS, D. D., AND JONES, K. S. Natural language processing for information retrieval. *Communications of the ACM 39*, 1 (1996), 92–101.

[67] LU, Z. Scalable distributed architectures for information retrieval. Tech. Rep. UM-CS-1999-049, , 1999.

[68] LU, Z., AND MCKINLEY, K. S. Searching a terabyte of text using partial replication. Tech. Rep. UM-CS-1999-050, , 1999.

[69] LU, Z., AND MCKINLEY, K. S. Partial collection replication versus caching for information retrieval systems. In *Research and Development in Information Retrieval* (2000), pp. 248–255.

[70] MACLEOD, I. A., MARTIN, P., AND NORDIN, B. A design of a distributed full text retrieval system. In *Proceedings of the 9th annual international ACM SIGIR conference on Research and development in information retrieval* (1986), ACM Press, pp. 131–137.

[71] MANBER, U., AND MYERS, G. Suffix arrays: a new method for on-line string searches. *SIAM Journal of Computing 22*, 5 (Oct. 1993), 935–948.

[72] MANNING, C., AND SCHTZE, H. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[73] MARKATOS, E. P. On caching search engine query results. *Computer Communications 24*, 2 (2001), 137–143.

[74] MCBRYAN, O. A. Genvl and wwww: Tools for taming the web. In *Proceedings of the First International World Wide Web Conference (WWW1)* (1994).

[75] MELNIK, S., RAGHAVAN, S., YANG, B., AND GARCIA-MOLINA, H. Building a distributed full-text index for the web. In *World Wide Web* (2001), pp. 396–406.

[76] MENG, W., LIU, K.-L., YU, C. T., WU, W., AND RISHE, N. Estimating the usefulness of search engines. In *ICDE* (1999), pp. 146–153.

[77] MOFFAT, A., AND ZOBEL, J. Parameterised compression for sparse bitmaps. In *Research and Development in Information Retrieval* (1992), pp. 274–285.

[78] MOFFAT, A., AND ZOBEL, J. Fast ranking in limited space. In *ICDE* (1994), pp. 428–437.

[79] MOFFAT, A., AND ZOBEL, J. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems 14*, 4 (1996), 349–379.

[80] MOFFAT, A., ZOBEL, J., AND SACKS-DAVIS, R. Memory efficient ranking. *Information Processing and Management 30*, 6 (1994), 733–744.

[81] MOHRI, M. Minimization algorithms for sequential transducers. *Theoretical Computer Science 234*, 1–2 (2000), 177–201.

[82] NAVARRO, G., BAEZA-YATES, R. A., BARBOSA, E. F., ZIVIANI, N., AND CUNTO, W. Binary searching with nonuniform costs and its application to text retrieval. *Algorithmica 27*, 2 (2000), 145–169.

[83] NETCRAFT. Netcraft web server survey.

[84] ONLINE COMPUTER LIBRARY CENTER. Size and growth. wcp.oclc.org/wc/stats/size.html.

[85] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The pagerank citation ranking: Bringing order to the web. Tech. rep., Stanford Digital Library Technologies Project, 1998.

[86] PINKERTON, B. Webcrawler. finding what people want. Tech. rep., 2000.

[87] PLANET, B. The deep web: Surfacing hidden value. Tech. rep., Whitepaper, 2000.

[88] RADEV, D. R., AND MCKEOWN, K. Generating natural language summaries from multiple on-line sources. *Computational Linguistics 24*, 3 (1998), 469–500.

[89] RIBEIRO-NETO, B. A., AND BARBOSA, R. A. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the third ACM conference on Digital libraries* (1998), ACM Press, pp. 182–190.

[90] RICHARD SELTZER, ERIC J. RAY, D. S. R. *The AltaVista Search Revolution*. Osborne McGraw-Hill, 1997.

[91] RISVIK, K. M. A search processor and method for retrieval of data and the usage in a search engine. International Patent PCT/NO99/00233, 1999.

[92] RISVIK, K. M., AASHEIM, Y., EGGE, T., AND PETTERSEN, H. Search engine with hierarchically stored indices. International Patent Pending, 2003.

[93] RISVIK, K. M., AASHEIM, Y., AND LIDAL, M. Multi-tier architecture for web search engines. In *Proceedings of the first Latin-american Web Conference* (Santiago, Chile, 2003), p. accepted.

[94] RISVIK, K. M., AND EGGE, T. The FAST search engine kernel and its performance characteristics. Tech. rep., Fast Search & Transfer ASA, 2002.

[95] RISVIK, K. M., EGGE, T., AND HALAAS, A. The FAST distributed processing architecture (DPA). Tech. rep., Fast Search & Transfer ASA, 2002.

[96] RISVIK, K. M., EGGE, T., SVINGEN, B., AND HALAAS, A. Search engine with two-dimensional linear scalable parallel architecture. International Patent PCT/NO99/00155, 2000.

[97] RISVIK, K. M., AND MICHELSEN, R. Search engines and web dynamics. *Computer Networks (Amsterdam, Netherlands: 1999) 39*, 3 (2002), 289–302.

[98] Risvik, K. M., Mikolajewski, T., and Boros, P. Query segmentation for web search. In *Proceedings of the Twelfth International World Wide Web Conference (WWW2003)* (2003).

[99] Salton, G. *Automatic Text Processing: The Transformational, Analysis, and Retrieval of Information by Computer.* Addison-Wesley, Reading, Massachusetts, U.S.A., 1989.

[100] Silverstein, C., Henzinger, M., Marais, H., and Moricz, M. Analysis of a very large altavista query log. Tech. Rep. 1998-014, Digital SRC, 1998. http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-1998-014.html.

[101] Silverstein, C., Henzinger, M., Marais, J., and Moricz, M. Analysis of a very large altavista query log. Tech. rep., SRC Technical Note, 1998.

[102] Spink, A., Ozmutlu, H., Ozmutlu, S., and Jansen, B. U.s. versus european web searching trends. In *Proceedings of SIGIR-02, 25th ACM International Conference on Research and Development in Information Retrieval* (2002).

[103] Strzalkowski, T. *Natural Language Information Retrieval.* Kluwer Academic Publishers, 1999.

[104] Suel, T., Mathur, C., Wu, J., Zhang, J., Delis, A., Kharrazi, M., Long, X., and Shanmugasunderam, K. Odissea: A peer-to-peer architecture for scalable web search and information retrieval, 2003.

[105] Sullivan, D. Avoiding the search gap.

[106] Talim, J., Liu, Z., Nain, P., and Jr., E. G. C. Controlling the robots of web search engines. In *SIGMETRICS/Performance* (2001), pp. 236–244.

[107] Tomasic, A., and Garcia-Molina, H. Caching and database scaling in distributed shared-nothing information retrieval systems. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data* (1993), ACM Press, pp. 129–138.

[108] Tomasic, A., and Garcia-Molina, H. Query processing and inverted indices in shared: nothing text document information retrieval systems. *The VLDB Journal 2*, 3 (1993), 243–276.

[109] TURTLE, H., AND CROFT, W. B. Inference networks for document retrieval. Technical Report UM-CS-1990-007, University of Massachusetts, Amherst, Computer Science, Mar. 31, 1990.

[110] UKKONEN, E. On-line construction of suffix trees. *Algorithmica 14*, 3 (1995), 249–260.

[111] VAN RIJSBERGEN, C. J. *Information Retrieval.* Butterworths, 1979.

[112] WILLIAMS, H. E., ZOBEL, J., AND ANDERSON, P. What's next? - index structures for efficient phrase querying. In *Proceedings of the Tenth Australasian Database Conference* (1999).

[113] WITTEN, I. H., MOFFAT, A., AND BELL, T. C. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[114] XIE, Y., AND O'HALLARON, D. Locality in search engine queries and its implications for caching, 2002.

[115] ZOBEL, J., AND MOFFAT, A. Adding compression to a full-text retrieval system. *Software – Practice and Experiment 25*, 8 (1995), 891–903.

[116] ZOBEL, J., MOFFAT, A., AND RAMAMOHANARAO, K. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst. 23*, 4 (1998), 453–490.

A Publications

- **Search Engines and Web Dynamics**. Knut Magne Risvik and Rolf Michelsen, In Computer Networks,Volume 39, Number 3, 2002.
- **The FMS Search kernel and its performance characteristics**. Knut Magne Risvik and Tor Egge, Technical Note, 2002.
- **The FAST DPA**. Knut Magne Risvik, Børge Svingen, Arne Halaas and Tor Egge, Technical note, 2002. Also submitted to IEEE Transaction of Knowledge and Information Systems.
- **Multi-tier Architecture for Web Search Engines**. Knut Magne Risvik, Yngve Aasheim and Mathias Lidal, In Proceedings of the first Latin-American Web Conference, LA-WEB 2003, Santiago, Chile.
- **Linguistics in Large-scale Web Search Engines**. Jon Atle Gulla, Per Gunnar Auran and Knut Magne Risvik, In Proceedings of the Natural Language Processing and Information Systems, 6th International Conference on Applications of Natural Language to Information Systems, NLDB 2002, Stockholm, Sweden.

B Patents

Note: These patents are not enclosed in this print of the thesis

- **A Search processor and method for retrieval of data and the usage in a search engine**, Knut Magne Risvik, International Patent PCT/NO99/00233, 1999.
- **Search Engine with two-dimensional linear scalable parallel architecture**. Knut Magne Risvik, Tor Egge, Børge Svingen and Arne Halaas, International Patent PCT/NO99/00155, 2000.
- **Search Engine with Hierarchically Stored Indices**. Knut Magne Risvik, Yngve Aasheim, Tor Egge and Håvard Pettersen. International Patent pending, 2003.

Search Engines and Web Dynamics

Knut Magne Risvik
Overture Services AS
P.O.Box 4452 Hospitalsløkkan
NO-7418 Trondheim, Norway
knut.risvik@overture.com

Rolf Michelsen
Fast Search & Transfer ASA
P.O.Box 4452 Hospitalsløkkan
NO-7418 Trondheim, Norway
rolf.michelsen@fast.no

Abstract

In this paper we study several dimensions of web dynamics in the context of large-scale Internet search engines. Both growth and update dynamics clearly represent big challenges for search engines. We show how the problems arise in all components of a reference search engine model.

Furthermore, we use the FAST Search Engine architecture as a case study for showing some possible solutions for web dynamics and search engines. The focus is to demonstrate solutions that work in practice for real systems. The service is running live at **www.alltheweb.com** and major portals worldwide with more than 30 million queries a day, about 2.1 billion documents, a crawl base of more than 5 billion documents, updated every 11 days, at a rate of 400 documents/second.

We discuss future evolution of the web, and some important issues for search engines will be scheduled and query execution as well as increasingly heterogeneous architectures to handle the dynamic web.

1 Introduction

Search Engines have grown into by far the most popular way for navigating the web. The evolution of search engines started with the static web and relatively simple tools such as WWW [17]. In 1995 AltaVista launched and created a bigger focus on search engines [19]. The marketplace for search engines is still dynamic, and actors like FAST (www.alltheweb.com), Google, Inktomi and AltaVista are still working on different technical solutions and business models in order to make a viable business, including paid inclusion, paid positioning, advertisements, OEM searching, etc.

A large number of analysis have been made on the structure and dynamics of the web itself. Conclusions are drawn that the web is still growing at a high pace, and the dynamics of the web is shifting. More and more dynamic

and real-time information is made available on the web. The dynamics of the web creates a set of tough challenges for all search engines.

In Section 2 we define a reference model for Internet search engines. In Section 3 we survey some of the existing studies on the dynamics of the web. Our focus is on the growth of the web and the update dynamics of individual documents on the web. In Section 4 we provide an overview of the FAST Crawler and describe how its design meets the challenges of web growth and update dynamics. We continue in Section 5 with a similar description of the indexing and search engines. Finally, we outline some future challenges and provide some benchmarking figures in Section 6 and Section 7, respectively.

The FAST Search Engine technology is used as a case study throughout the paper. The focus of the paper is on how web dynamics pose key challenges to large-scale Internet search engines and how these challenges can be addressed in a practical, working system. The main contribution of this paper is to offer some insight into how a large-scale, commercially operated Internet search engine is actually designed and implemented.

2 A Search Engine Reference Model

Most practical and commercially operated Internet search engines are based on a centralized architecture that relies on a set of key components, namely Crawler, Indexer and Searcher. This architecture can be seen in systems including WWW [2], Google[5], and the FAST Search Engine [1], and can be illustrated in Figure 1.

Definition 5 Crawler. *A crawler is a module aggregating documents from the World Wide Web in order to make them searchable. Several heuristics and algorithms exist for crawling, most of them based upon following links in hypertext documents.*

Definition 6 Indexer. *A module that takes a collection of documents or data and builds a searchable index from them. Common methods are inverted files, vector spaces, suffix structures and hybrids of these.*

Definition 7 Searcher. *The searcher is working on the output files from the indexer. The searcher accepts user queries from the dispatcher (defined below), executes the query over its part of the index, and returns sorted search results back to the dispatcher with document ID and the relevance score (defined below).*

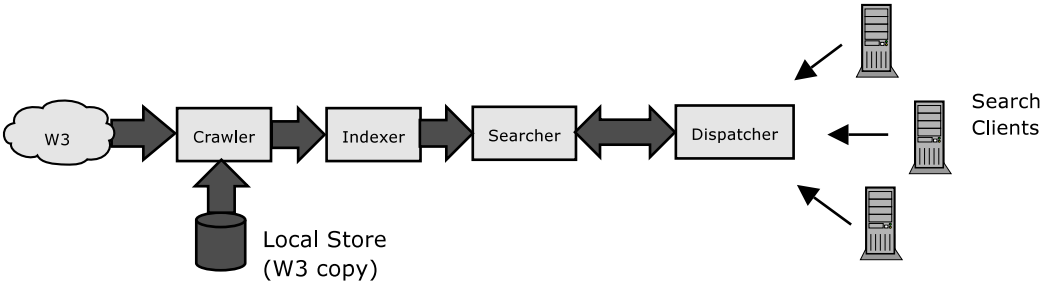


Figure 1. Search Engine reference model

Definition 8 Dispatcher. *The dispatcher receives the query from the user, compiles a list of searchers to execute the query, sends the query to the searchers and receives a sorted list of results back from each searcher. For each result it receives a unique document ID, and the relevance score. The hits from the searchers are then merged to produce the list of results with the highest relevance scores for presentation to the user.*

Some systems also keep a local store copy of the crawled data.

Definition 9 Local Store *A local store copy is a snapshot of the web at the given crawling time for each document.*

Systems usually run the crawler, indexer, and searcher sequentially in cycles. First the crawler retrieves the content, then the indexer generates the searchable index, and finally, the searcher provides functionality for searching the indexed data. To refresh the search engine, this indexing cycle is run again.

In real systems, the different phases of the indexing cycle may overlap, be split into different sub-phases, and so on. For instance, it is unacceptable to stop the searcher while running the crawler or indexer. However, in most search systems these fundamental steps of the indexing cycle are easily discernible.

The literature often distinguishes between *batch* and *incremental* crawlers. (Different papers use different terms and definitions. This paper uses the definitions by Cho and Garcia-Molina [7].) A batch crawler starts from scratch with an empty local store in every indexing cycle, and it never fetches the same document twice in that cycle. An incremental crawler never erases its local store. When it has retrieved a sufficient number of documents, it continues by re-fetching documents and updating the local store copies. When

a new indexing cycle starts, an incremental crawler continues where it left off when the last indexing phase started.

3 The Dynamics of the Web

In this section we outline the nature of web dynamics. We define the different aspects of web dynamics, and we review the literature on the topic. We do not attempt to provide a complete review of the published studies but rather focus on a number of representative and significant works.

3.1 Dimensions of Web Dynamics

The concept of web dynamics has several dimensions and all are important for large-scale Internet search engines. First, the web grows in size, and most studies agree that it grows at an exponential rate. This poses a serious challenge of scalability for search engines that aspire to cover a large part of the web.

Second, the content of existing web documents is updated. A search engine must not only have good coverage, but perhaps even more important, the local store copy must be fresh. A search engine must not only scale to a large volume of documents, but it must also be able to refresh these documents in a timely manner.

This requirement will only grow more important as people start turning to search engines for breaking news and other dynamic content.

Furthermore, the link structure of the web is in constant flux. Links between documents are constantly being established and removed. The dynamics of the link structure is important for search engines as long as link structure is an important component in ranking of search results. It is likely that using link structure in ranking creates a slow-working positive feedback loop in the entire web to make popular sites even more popular at the expense of less known or new sites.

A number of studies of link structure have been made, for instance the recent study by Broder et.al [10]. However, none of these studies cover the dynamics of the link structure. We do not consider dynamics of the link structure in this paper.

Yet another dimension of web dynamics is introduced with structure. XML is evolving rapidly in inter-service systems, and the ability for a search engine to get under the hood of a presentation engine to understand the structure and semantics of the data is a key feature for the next generation engines.

3.2 Web Growth Dynamics

The web has grown at an incredible rate since the very inception about ten years ago. The web started out with a one-to-one connection between the data to be made available and the HTML page used to present the information. The web has been the driving factor for establishing a new era for storage and retrieval of information, and we are seeing the beginning of yet another. The web took storage from the application era into the information era, and the web is actually a huge storage and retrieval network.

HTML is a format description for documents. The combination of URIs and HTML made a connection between a file system object and an URI obvious, and web servers are today most commonly applications that serve HTML files directly from a file system upon requests. However, the enormous growth in information that we want to publish on the web has created the need and space for more advanced publication systems tying business applications to web servers.

The web being the platform for a new information era has caused the web to evolve into an application and transactional space as well. E-trading is becoming a major application on the web.

These two evolutionary trends on the web have erased the connection between HTML documents, URIs, and the actual content being presented. This has furthermore limited the percentage of the web that is actually indexable by a search engine. E-trading and advanced information systems introduce personalized or transaction-dependent content that is not normally accessible by standard web aggregation methods.

Several studies of the size of the web have been conducted. OpenText conducted a very early study in late 1995 [3]. It showed about 223,000 unique web servers, and the number of documents estimated to be 11.4 million.

In 1998 and 1999, Lee Giles and Steve Lawrence conducted well-known tests to estimate the size of the web[15], and to explore the accessibility of the indexable web[16].

In [15] the study is based upon multivariate analysis of the search engine coverage and overlap to estimate the size of the web. Six search engines were used, and the lower bound estimate for the size of the web was 320 million pages. Since the study is based on search engines and the result sets overlap, the measure is clearly on the indexable web, not taking into account the percentage of the web that is not touched by any search engine. Looking at the estimated coverage of each search engine with respect to the combined coverage, HotBot was the most comprehensive at the time of that measure (approximately 57% coverage).

The test was repeated and extended in Nature in 1999 [16]. The 11 months that had passed showed a significant increase in the number of indexable documents found. The lower bound of size was estimated to 800 million documents, and the search engines had significant less coverage of the indexable web. The maximum coverage estimated was 16

In [13], a theory for the growth dynamics of the World Wide Web is presented. Two stochastic classes are considered, namely the growth rates of pages per site, and the growth of new sites. A universal power law is predicted for the distribution of the number of pages per site. The paper brings theories that enable us to determine the expected number of sites of any given size without extensive crawling.

These three papers discuss the growth in what is referred to as the "indexable" web, but no study was performed of the percentage of pages "indexable" versus "non-indexable". A study by Bright Planet LLC [18] introduced the concept of the "Deep Web".

The deep web is easily identified as the subset of the web not discussed in [15] and [16], the "non-indexable" web. The percentage of web pages belonging to the non-indexable category is growing at a much higher rate than the indexable pages. This is a natural cause of the web moving from a simple document share space into an information sharing space and even into an application sharing space.

Key findings are in the study by Bright Planet are:

- 7500 terabytes of information (19 terabytes assumed to be the surface web).
- Approximately 550 billion documents.
- Largest growing category of web information.
- Highly relevant content found in the deep web.

3.3 Document Update Dynamics

A number of challenges must be faced to handle document update dynamics in a large-scale Internet search engine. First, we must develop a model for how documents are updated. Then, we must develop a crawling strategy that maximizes the freshness of the local store given this document update model. To evaluate the performance of various update strategies, we need mechanisms for measuring the freshness of the local store.

These challenges have been studied in the existing literature. Cho and Garcia-Molina have published several studies both on how web documents

are updated and on crawling strategies [8] [7] [6]. Their models and experiments indicate that web document updates can be modeled as independent Poisson processes. That is, each document d_i is updated according to a Poisson process with change rate l_i, and the change rates are independent. Their experiments on the web indicate that an average document is changed once every ten days and that 50% of all documents are changed after 50 days [7].

Cho and Garcia-Molina have also developed statistical estimators for the Poisson parameters under various assumptions. They have derived estimators for uniform and random observation of the web documents, and for known and unknown time of last document update. The new estimators are much better than the naive estimator: the number of document updates observed divided by the observation time[6].

Finally, they have also presented an optimal crawling strategy given their model of a crawler and local store. They also propose a framework for measuring how up-to-date the local store is through their concepts of freshness and age. They define the freshness of a document d_i at time t as the probability that the document is up-to-date at the given time. Age is defined to be the 0 for documents that are up-to-date and the time since the last document update in the real world for others. For both freshness and age, the interesting measure is the average freshness or age both over all documents and over time[8]. In this paper we will use the term "freshness" in a more informal manner.

Based on their models for document change and freshness measures, they present some optimal scheduling policies for an incremental crawler. They make the following observations:

- Refreshing document using uniform update frequencies is always better than using document update frequencies that are proportional to the estimated document change frequencies l_i.

- The scheduling policy optimizing freshness penalizes documents that are changed too often. Intuitively, these documents are likely to change again very soon and hence do not contribute much to the overall freshness of the local store.

- The scheduling policy optimizing age favors documents that are changed very often, but the actual change is small.

Brewington and Cybenko[4] also performed a number of experiments to discover how web documents are updated. They also conclude that web

documents are updated according to a Poisson process. Brewington and Cybenko also propose a novel measure of freshness, termed (a, b)-currency. A document d_i is (a, b)-current if it were up-to-date b time units ago with probability a. This measure captures both the aspirations and the actual achievements regarding freshness. A daily newspaper may be (0,95, 1 day)-current, meaning that 95% of all articles in the paper was up-to-date one day ago. They estimate that an Internet search engine containing 800 million documents must refresh 45 million documents every day to be able to maintain (0,95, 1 week)-currency.

Edwards, McCurley, and Tomlin [9] present a crawler that minimizes the number of obsolete documents in the repository without making any a priori assumptions about how documents are updated. They use measured document change rates and divide their crawling resources on documents according to their change rate. They solved a vast optimization problem to find the optimal distribution of crawling resources.

3.4 Search Engines and Search Technology

There is not a rich body of literature describing practical large-scale crawling and searching systems, and in particular, very few address issues relating to the dynamic web. Brin and Page have described an early version of the Google system [5]. They address issues relating to growth of the web and scaling of the document volume, but they do not address refreshing of the local store.

Heydon and Najork describe their scalable web crawler [11]. Their crawler is scalable in the sense that the required machine resources are bounded and do not depend on the number of retrieved documents. Their crawler reportedly runs on a single, large machine, but it can probably quite easily be implemented in a crawler cluster to scale with even larger document volumes or processing load in a manner similar to our FAST Crawler as discussed later in this paper. Heydon and Najork do not discuss refreshing the crawled documents.

Edwards, McCurley, and Tomlin[9] provide some design details about their crawler. This crawler runs incrementally, constantly refreshing documents to ensure freshness over time, as described in the previous section. They also provide some details about their scheduling algorithm. Each document is classified according to its measured update frequency, and crawling resources are then divided among these classes. They formulated and solved a vast optimization problem for computing how to allocate the crawling resources among the different document update classes.

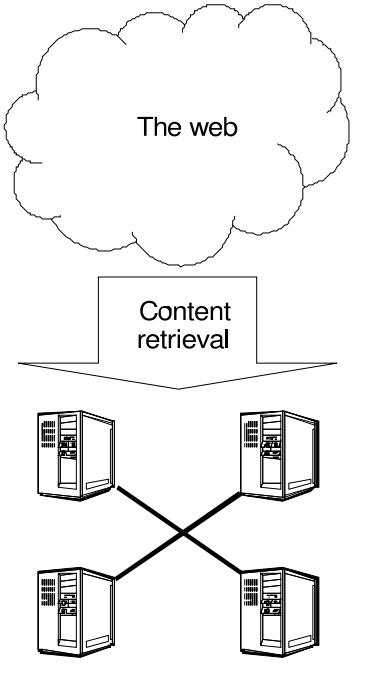


Figure 2. FAST Crawler deployment overview

4 Aggregation of Dynamic Content

In this section, we use the FAST Crawler as a case study to illustrate how we have addressed the challenges of scaling with the size of the web and ensuring the freshness of our local store.

4.1 Overview of the FAST Crawler

The FAST Crawler consists of a cluster of interconnected machines as depicted in Figure 2. Each machine in this cluster is assigned a partition of web space for crawling. All crawler machines communicate with all other machines in a star network. However, the machines work relatively independent of each other, only exchanging information about discovered hyperlinks.

Figure 3 depicts the main components of a single crawler machine in our system. The solid arrows indicate flow of document content, and the dotted arrows indicate flow of document URIs, possibly with associated meta-information.

The Document Scheduler is responsible for figuring out which document

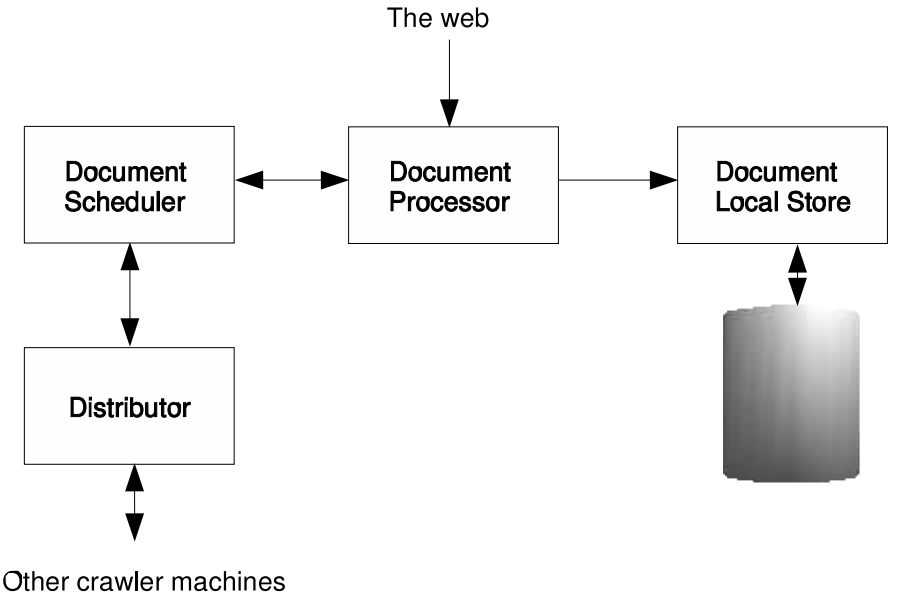


Figure 3. Components and data flow in a single crawler machine

to crawl next. Conceptually, this means maintaining a huge prioritized queue of URIs to be crawled while also observing the informal "social code" of web robots such as the robots exclusion protocol [14] and various constraints on the access pattern to individual web servers imposed to avoid overloading servers with crawler requests. The Document Scheduler sends a stream of URIs to the Document Processor.

The Document Processor is responsible for retrieving documents from web servers and performing any processing on them. This component consists of a manager module that provides a plug-in interface used by a set of processing modules. The manager routes documents through a pipeline of processing modules based on configuration information, the document content type, etc. This architecture provides the necessary flexibility to quickly support new document processing functionality in the crawler. We currently have processing modules for parsing HTML documents and various multimedia formats, classification of language, classification of offensive content, etc. It is the HTML parser that discovers hyperlinks to other documents and passes these back to the scheduler.

After processing, the crawler stores relevant document content and meta-information in the Local Store. Document content represents the bulk of the data, and we have optimized our storage system for efficiently writing

or updating individual documents and for streaming the entire document collection to the indexing step. There is no need for efficient random access reading of document content. We use a hybrid of a hashing and logging storage system. Hirai et al describe the WebBase system, a system that is very similar to our storage system[12]. Some document meta-information is kept in a high-performance, random access database.

The Distributor exchanges hyperlink information with other machines in the crawler cluster. There is a static mapping from the relevant hyperlink information to a crawler machine.

The crawler architecture also contains two important components for duplicate detection and link-based ranking, respectively. These modules are relatively complex and outside the scope of this paper.

The FAST Crawler is an incremental crawler. That is, when the document repository has reached its target size, the crawler continues by refreshing existing documents to detect changes. It fetches new documents only to replace documents that are deleted from the repository because the crawler discovers that they have been deleted from the web or for other reasons. The crawler is never stopped. It is temporarily suspended during a part of the indexing process, but when indexing is completed the crawler resumes operation where it left off.

4.2 Scalability

A large-scale crawler must be scalable both with respect to *document storage capacity* and *document retrieval capacity*. In our architecture, each crawler machine is responsible for all retrieval, processing, and storage for a partition of the document space. The different machines constituting a crawler cluster work independently except for exchanging discovered URIs. Hence, the storage capacity, C_S, and the processing capacity, C_P, of the crawler cluster is the sum of the capacity of each individual machine:

$$C_S = \sum_j C_{S,j} \quad (1)$$

$$C_P = \sum_j C_{P,j} \quad (2)$$

An additional constraint on the crawler retrieval and processing capacity is the capacity defined by the total network bandwidth available to the crawler cluster, C_N. Hence, the total retrieval capacity of the crawler cluster is:

$$C_R = min(C_P, C_N) \tag{3}$$

Usually, network bandwidth represents the highest cost for running a large-scale crawler and hence it makes sense to dimension the system so that $C_N \leq C_P$. Note that the bandwidth used for communicating internally in the crawler cluster is proportional to the inbound bandwidth used for retrieving content from the web. The internal bandwidth is only used for exchanging hyperlink information, and the number of hyperlinks is proportional to the number of retrieved documents.

We can easily increase the document storage capacity, C_S, by adding new machines to the crawler cluster and redefine the workload partitioning in the distributor accordingly. This will also give us some extra processing power, C_P, "for free". Increasing only the document storage capacity does not require more network bandwidth, C_N, between the crawler and the web or internally between the individual crawler nodes comprising the cluster. As a result, the crawler scales linearly with document storage capacity.

We also scale linearly with document retrieval capacity, C_R. Retrieving, processing, and storing more documents per unit of time requires a linear increase in inbound network bandwidth from the web and also a linear increase in network bandwidth between the machines comprising the crawler cluster, C_N.

Scaling document retrieval capacity by increasing network capacity assumes that each machine in the crawler cluster has enough spare processing capacity to handle the increased load, that is . If this is not the case, then additional machines must be added to the cluster to achieve the required processing power. This will also increase the document storage capacity. Hence, the total system scales linearly with both storage and retrieval capacity as long as there is a reasonable balance between the two. In practice, this is usually the case, as we do not want the ratio of retrieved documents per unit of time to the total number of documents to drop as we scale the system, as this will eventually impact our freshness. For a crawler to maintain a given freshness, F, the following ratio must be kept constant while scaling (let k be any constant):

$$k\frac{C_R}{C_S} = F \tag{4}$$

A system that scales to many components must also be robust with regards to failure of any of these components or otherwise the probability of having a working system will drop with the number of components. Our

crawler is robust against failure of any machine in the crawler cluster. Each machine works independently on scheduling, retrieving, processing, and storing documents. The only dependency between machines is the exchange of information about new hyperlinks. Hyperlink information for an unavailable crawler machine is simply queued on the sending machine until the designated receiver again becomes available.

4.3 Freshness through Scheduling

The FAST Crawler provides fresh data to the search engine through its very high document retrieval capacity and its scheduling algorithm prioritizing retrieval of documents most likely to have been updated on the web. A good scheduling algorithm increases the constant factor k in the freshness equation in the previous section.

In normal operation, the crawler is refreshing a local store of approximately a constant number of documents limited by the capacity of the search engine and the crawler itself. In this state, the crawler will only retrieve new documents when old documents are removed from the local store either because the crawler attempted a refresh but the document does not exist on the web anymore or for other reasons. When the document storage capacity is increased, we normally relatively quickly retrieve enough documents to again reach this steady state.

Providing fresh search results to users implies short indexing cycles, and we do not have the capacity to refresh all documents in our local store between each indexing cycle. In this situation the scheduling algorithm is a key element in ensuring a fresh local store. To maximize freshness, we must spend as much as possible of our crawler network capacity on refreshing documents that have actually changed.

The FAST Crawler currently uses a relatively simple algorithm for adaptively computing an estimate of the refresh frequency for a given document. Basically, this algorithm decreases the refresh interval if the document was changed between two retrievals and increases it if the document has not changed. This is used as input to the scheduler, which prioritizes between the different documents and decides when to refresh each document. In addition, the scheduler is configured with global minimum and maximum refresh intervals to keep the refresh rate for a document within sensible bounds, e.g. to allow refreshing of documents for which we have never observed a change.

Cho and Garcia-Molina observe that it is not always optimal to refresh documents that are updated very frequently "as often as possible [8] [6]. Intuitively, these documents will always be obsolete anyway when the repos-

itory is indexed. In fact, they conclude that a uniform refresh policy where all documents are updated equally often is always superior to such a proportional refresh policy. In practice, this is not a big problem. With a repository of the size required for a large-scale search engine, there are always enough documents waiting to be refreshed to diminish the effect of the relatively few documents that are updated very often. Also, we configure the scheduler to avoid rescheduling any document more than once for each indexing cycle.

4.4 Freshness through Heterogeneity

When optimizing freshness through scheduling, we assume that all documents have the same freshness requirements. Having a fresh copy of one document in the repository is just as valuable as having a fresh copy of any other document. This is not always the situation in the real world. Consider the following two examples:

- There is an industry trend to offer a "pay for inclusion" service to content providers. This offer usually comes with service level guarantees covering refresh and indexing intervals.

- There is a trend towards searching in increasingly dynamic content, e.g. news. This content must be refreshed and indexed very often to be of any value to users.

The above two scenarios cannot be supported in a large-scale crawler using a relatively simple scheduling algorithm alone. In the FAST Crawler, we have solved this problem by permitting heterogeneity in the cluster of crawler machines. This means that the different machines can be configured with very different storage and processing capacity, $C_{S,i}$ and $C_{P,i}$. A relatively small number of machines are dedicated to special purposes, such as crawling of content from paid inclusion programs or news sites. These machines can be configured specially with the service requirements of these services in mind, and we can control the load on these machines without sacrificing the high capacity and efficiency of the main bulk of the machines in the crawler cluster.

We still keep these dedicated machines as a part of the cluster just as all the other machines so that all machines can efficiently share link information, computed link ranks, etc. The cost is relatively small - only a small increase in the complexity of the distributor.

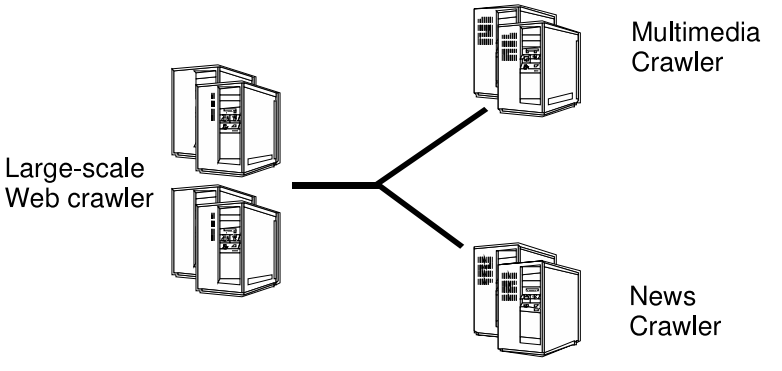


Figure 4. Example of heterogeneous crawler deployment with sub-clusters for different crawling

Figure 4 shows an example of a heterogeneous crawler cluster. This figure depicts a crawler cluster with a few dedicated machines for crawling news content in addition to the large-scale web crawler. We have also included a few machines comprising a multimedia crawler to illustrate the flexibility of the architecture. In this example, the large-scale web crawler will typically be configured with a large C_S but a relatively low F. The news crawler will be configured with a lower C_S to maintain a high F. The multimedia crawler will have high C_S and low F just as the large-scale web crawler, but operating this crawler as a separate part of the cluster allows control over resources used for different media types.

4.5 Cooperation with Providers

Another approach, that we are experimenting with, is cooperating with content providers to further improve freshness. There are different models for cooperating with content providers, and in this section we outline the different options.

InfoSeek once proposed a standard for a web server meta-file named *siteinfo.txt* to complement the more established robots.txt de facto standard. The purpose of the siteinfo.txt file was to provide information to crawlers about which documents had changed, mirrors of the server, etc. The siteinfo.txt standard was never able to establish itself and our own crawling indicates that today no server is using it. Today, a number of content providers use proprietary meta-files to publish information about their sites

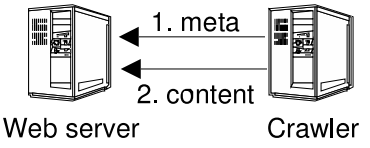


Figure 5. Meta-information pull model

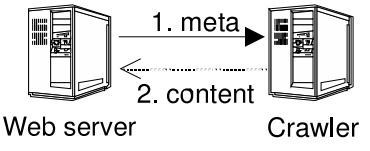


Figure 6. Meta-information push model

and how they are updated.

This is a pull model that permits the crawler to perform efficient per-site scheduling. In this model, the crawler periodically fetches some meta-information from the web server. It uses this information to influence its own scheduler with hints about new or changed documents. This model is illustrated in Figure 5. (The arrowheads indicate the direction of the requests.) However, building a crawler that supports all these proprietary meta-information formats is a daunting task.

A more elaborate approach is to push content or meta-information directly from the content provider to the crawler. In this model the content provider sends a notification to the crawler whenever a document is added, modified, or deleted on a site. In its simplest form, the content provider just submits the URI of the document. This method can be enhanced by also submitting various meta-information, or even the entire content of the documents. However, care must be taken to avoid opening the crawler and search systems to spamming and other abuse so such a service can only be offered to partners. This model is illustrated in Figure 6 with the dotted arrow indicating an optional request.

A third approach is a hybrid between the two above. There are many obstacles to having push technology deployed at a large set of content providers. For instance, site operators may be reluctant to increase the complexity of their systems by installing additional software components. The hybrid ap-

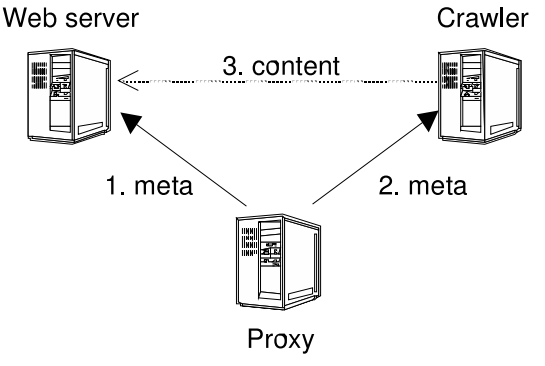


Figure 7. Hybrid meta-information pull-push model

proach involves developing and deploying special proxy systems that read site meta-information files, act according to the instructions in those files, and then use push technology to update the crawler. These systems use a single interface to the crawler but are otherwise independent of the crawler, and hence these proxies can be developed and deployed independently. This is illustrated in Figure 7.

The hybrid model is quite powerful. The proxy may not limit itself to obtaining meta-information from only the server or set of servers being crawled. For instance, it can also fetch meta-information from other sources to discover "hot spots" in the web and then direct the crawlers to these spots.

After establishing the communication pipeline with the providers, a natural next step lies within better understanding the content of the provider. XML could be a significant player for describing both semantics and dynamics of the content.

5 Searching Dynamic Content

The second and third components in the reference search engine model, the Indexer and the Searcher, need also to handle the different dimensions of web dynamics. Traditionally, search engines have been based upon batch-oriented processes to update and build indexes. To handle the growth in size of the Web and the update dynamics, most traditional designs fall short. In this section, we will study several aspects and solutions for an indexer and

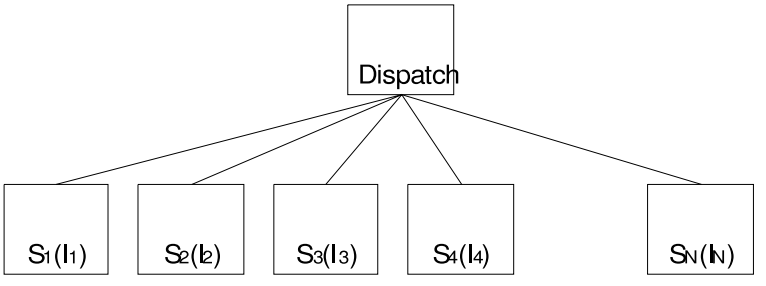


Figure 8. Linear Scaling with data size

a searcher to handle a dynamic web.

5.1 Scalability with Size and Traffic

Being able to handle the web growth calls for architectural solutions. Given traditional solutions with inverted files or equivalent, the cost of building, maintaining and searching an index is worse than linear.

One possible architecture for a Web search engine is the FAST Search Engine Architecture. It handles scalability in two dimensions, namely size and traffic volume. The architecture is a distributed architecture, and has two classes of nodes:

- **Search Nodes:** A Search node Si holds a portion of the index, Ii. The total index is $I = \bigcup_i I_i$ Each of the search nodes is a separate entity than holds a search kernel (searcher) that searches the index Ii and returns search results. The search nodes have no interconnection between them.

- **Dispatch Nodes:** A dispatch node does not hold any searchable data. The dispatcher is a routing, distribution, and collection/merging mechanism. A dispatch node receives queries and routes them to a set of underlying search nodes, $S_i \ldots S_j$. The results are collected and merged before they are sent to the issuing client.

A search node has two capacities, namely the number of documents on each node, $|D_i|$, and the query rate or traffic capacity, Ci. A dispatcher has one capacity, namely the dispatching capacity, C_di. The C_di depends on the number of search nodes the query is sent to.

Now, by using the two components described above, we can build a simple architecture to allow linear scaling with the data size. The architecture

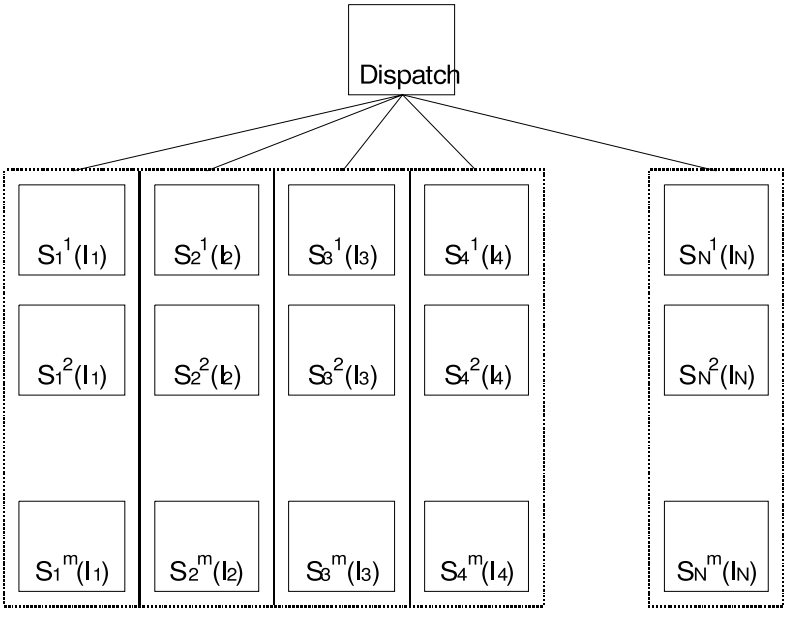


Figure 9. Scaling in size and capacity

is shown in Figure 8. Each box $S_i(I_i)$, is a search node holding the index partition I_i. The entire dataset, I, is partitioned out on the search nodes. The dispatch node is broadcasting queries to all search nodes in parallel and merging results from the search nodes to build the final result set.

Now, assuming that each search node handles the query individually of the other search nodes, we see that a linear scale up for increasing the overall size $|D|$ is achieved.

At the same time, to scale with the query rate, replication will provide the capacity mC_i. The dispatch node will know all search nodes in a column, and a round-robin algorithm can be used to rotate between the different columns. This extended architecture is illustrated in Figure 9. Here S_{ij} is a search node j handling partition i. As illustrated, the dispatcher knows all search nodes in each column, and can load-balance between them, achieving a linear performance scale up.

So, the number of search nodes required for a dataset I of size $|I|$ is derived as $\frac{|I|}{p}$, where p denotes the optimal size of a search node partition. This number is of course dependent on actual hardware configurations and costs. Furthermore, the number of search nodes to handle a given query rate Q_t, is derived as $\frac{Q_t}{C_i}$.

The limitation of this architecture clearly lies within the dispatching

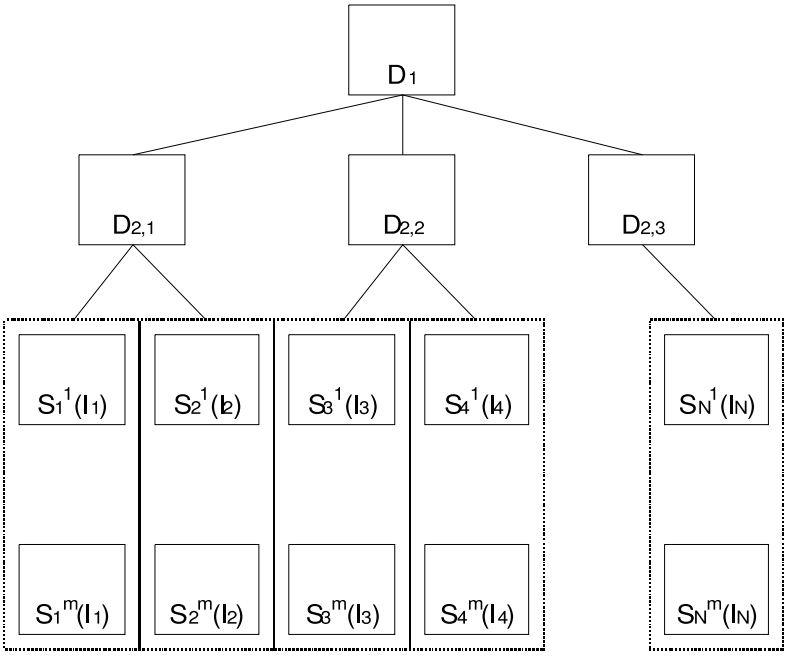


Figure 10. Multi-level dispatch architecture

system. A dispatcher has a capacity of merging results. Optimal merging algorithms are in order of $O(rlogm)$, where r denotes the number of sources and m the number of entries. Thus, the merge performance is limited by the number of search nodes that receive the query in parallel.

To ensure linear scalability, a multi-level dispatch system can be configured. By letting a dispatcher dispatch to a part of each row, and the letting a super-dispatcher dispatch to those dispatch nodes, we can use a tree-like architecture, as shown in Figure 10.

Any number of levels can be built to accommodate for the scale in the two dimensions. In the worst case, this will be a binary tree, where the number of nodes is $O(2N-1)$, and still linear to the size of the data being searched.

An immediate observation from the description above is that the architecture also has implicit fault-tolerance. By having multiple nodes with the same index partition, I_i, dispatchers can be fault-tolerant by detecting timeouts or non-replies. To ensure fault-tolerance on any level, the transparency of dispatcher/searcher can be utilized to have redundant dispatcher.

5.2 Handling Update Dynamics

The second dimension of web dynamics, the update dynamics, is also a problem facing search engines. Traditionally, structures used for indexing are based upon offline building, and in many cases this is a highly time consuming process.

There are two ways to make indexing processes more dynamic:

1. Identifying new inverse structures that allow for online updates.

2. Utilizing a heterogeneous architecture to allow for dynamic changes to the dataset.

For 1) there are several proposed solutions in the IR community, but none that has gotten big acceptance in the industry.

The FAST Search engine uses the second approach to cope with dynamic updates. Looking at the distributed architecture described above, the indexing is an easy goal for parallelization. Indexing D can be done by indexing $I_1, I_2, \ldots I_N$ individually. Since separate search nodes handle each partition, indexing can be done individually.

This solution still does not make indexing possible online on each search node, but by having slight overcapacity of search nodes, we can easily switch nodes on and off line to update parts of the dataset. Still, the system is

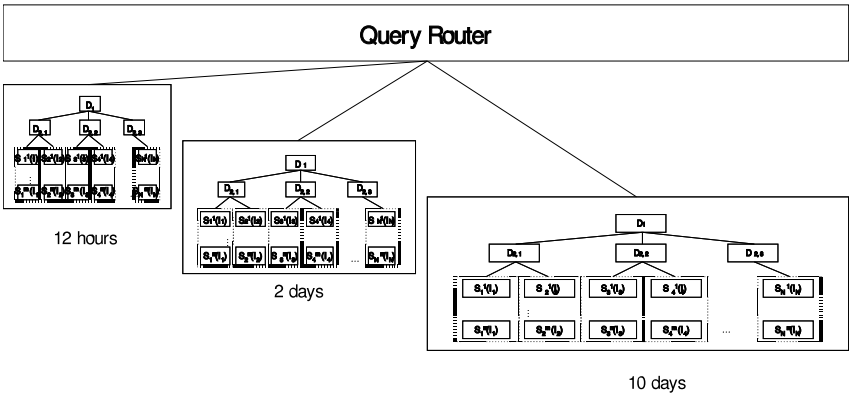


Figure 11. Multi-tier cluster

batch-oriented and the time from detecting a document update until it is pushed live will still be 20 hours or above.

By using a crawling system that allows heterogeneous clusters of crawler nodes, it is easy to identify sources with different update rates. Taking the architecture described in 5.1 and using that as a building block, we can have a cluster of search clusters where each cluster has a different update frequency.

A possible cluster solution is shown in Figure 11. In this example, we deploy three different clusters, each receiving a data feed from a different crawler cluster. The update frequencies of the clusters can then be different, but in general update cost (in either time or required hardware) is linear to the data size.

6 Future Challenges

The evolution of the dynamic web raises several significant challenges for search engines. First, the increasing dynamics and size makes intelligent scheduling increasingly important. Being able to update the most important parts of an index at a timely rate will be crucial in order to bring relevant search results to the users. Intelligent scheduling, heterogeneous crawling and push technology will be crucial to building aggregation and search systems capable of scaling with the web at a reasonable price.

The size of the web is clearly a big challenge, and one important question is arising: Do we actually need to search several billion documents for every query? Being able to intelligently classify and predict a probable subset of

the data set to search for given queries will enable us to build much more efficient and cost-effective.

At the same time, the "Deep Web" is most likely growing at a rate much higher than the current "indexable" web. There is no unified and practical solution to aggregate the deep web on a large scale, but push based technology and perhaps tight integration into publication and content management systems will evolve to address this challenge.

The explosive growth of the web also calls for more specific search engines. The introduction of focused crawling and document classification enables both crawlers and search engines to operate very efficiently within a topically limited document space. The Scientific search engines *scirus.com* and *www.researchindex.com* are good examples of engines that uses both focused crawling along with document classification. The dynamics also has a more homogeneous character within such a vertical, enabling a fresher search experience.

7 Conclusions

We have discussed several dimensions of web dynamics. Both growth and update dynamics clearly represent big challenges for search engines. We have shown how the problems arise in all components of a reference search engine model.

The FAST Search Engine architecture copes with several of these problems by its key properties. The overall architecture that we have described in this paper is quite simple and does not represent very novel ideas. The system architecture is relatively simple, and this makes it manageable even when it grows. In a real-life system with service level requirements, simplicity is crucial to operating the system and to being able to develop it further.

Being heterogeneous and containing intelligence with regards to scheduling and query processing makes this a real-life example of dealing with web dynamics issues today. The service running at www.alltheweb.com and major portals worldwide currently handle more than 30 million queries per day. Indexing happens every 11 days, and the full index size is currently about 700 million full-text documents. These documents were selected from a crawled base of 1.8 billion full-text documents. The crawler architecture enables us to crawl at rate of 400 documents/second and beyond.

The system is based on inexpensive off-the-shelf PCs running FreeBSD and our custom search software. We currently use approximately 500 PCs

for our production systems. Most of these machines are search nodes. We currently use 32 machines for crawling. The hardware configuration differs depending on the role the machine has in the architecture and the time of acquisition. Most machines are typically dual-Pentium machines with between 512 and 1024 Mbytes of memory.

Future evolution of web dynamics raises clear needs for even more intelligent scheduling to aggregate web content as well as technology for push-based aggregation. By doing more intelligent query analysis and processing, we will be able to do a sub-linear scaling with the growth of the web based on the ideas from Figure 11. It is possible to create a multi-tier system where one tier with few columns and many rows handles a relatively large part of the most popular queries. Another tier with more columns but fewer rows can then handle the remaining queries.

References

[1] Alltheweb. http://www.alltheweb.com.

[2] Html engine. http://www.cs.colorado.edu/home/mcbryan/Home.html.

[3] T. Bray. Measuring the web. In *Proceedings of the Fifth International World Wide Web Conference (WWW5)*, 1996.

[4] B. E. Brewington and G. Cybenko. How dynamic is the Web? *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):257–276, 2000.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th WWW Conference*, http://decweb.ethz.ch/WWW7/1921/com1921.htm, 1998.

[6] J. Cho and H. Garcia-Molina. Estimating frequency of change. Unpublished, 2000.

[7] J. Cho and H. Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proceedings of the Twenty-sixth International Conference on Very Large Databases*, 2000.

[8] J. Cho and H. Garcia-Molina. Syncronizing a database to improve freshness. In *Proceedings of the 2000 ACM International Conference on Management of Data (SIGMOD)*, 2000.

[9] J. Edwards, K. S. McCurley, and J. A. Tomlin. An adaptive model for optimizing performance of an incremental web crawler. In *World Wide Web*, pages 106–113, 2001.

[10] A. B. et. al. Graph structure in the web. In *Proceedings of the Ninth International World Wide Web Conference (WWW9)*, 2000.

[11] A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, 1999.

[12] J. Hirai, S. Raghavan, H. Garcia-Molina, and A. Paepcke. WebBase: a repository of Web pages. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):277–293, 2000.

[13] B. A. Huberman and L. A. Adamic. Evolutionary dynamics of the world wide web. Technical report, Xerox Palo Alto Research Center, 1999.

[14] M. Koster. A standard for robots exclusion.

[15] S. Lawrence and C. L. Giles. Searching the World Wide Web. *Science*, 280(5360):98–100, 1998.

[16] S. Lawrence and C. L. Giles. Accessibility of information on the web. *Nature*, 400(6740):107–109, 1999.

[17] O. A. McBryan. Genvl and wwww: Tools for taming the web. In *Proceedings of the First International World Wide Web Conference (WWW1)*, 1994.

[18] B. Planet. The deep web: Surfacing hidden value. Technical report, Whitepaper, 2000.

[19] D. S. R. Richard Seltzer, Eric J. Ray. *The AltaVista Search Revolution.* Osborne McGraw-Hill, 1997.

The FMS Search Kernel and its Performance Characteristics

Knut Magne Risvik
Yahoo!
P.O. Box 4452 Hospitalsløkkan
NO-7418 Trondheim, Norway
kmr@yahoo-inc.com

Tor Egge
Yahoo!
P.O. Box 4452 Hospitalsløkkan
NO-7418 Trondheim, Norway
tegge@yahoo-inc.com

Abstract

This paper describes the anatomy of the FMS Search engine, outlining data structures and algorithms. Key data structures are described in detail to show the novel elements for high-end performance. The search kernel is then analyzed to derive a performance model for the different aspects of searching, and to create a joint performance model which can be used for performance planning of large-scale engines. A sample document collection from the FAST Web Search service, and a real query log is then used to experimentally verify the performance model. We show that our search kernel will behave linear in document collection size given a certain set of constraints.

1 Introduction

A search engine kernel is a key part in all search engines and IR systems. The FMS Search Engine kernel (FMS Search) is the research variant of the search kernel component in all search products and services from FAST. A typical search engine installation consists of the following major modules:

- **Data aggregation and processing**. For a web search installation, this typically consists of a crawler and a chain of data processing (parsing, link extraction, deduping, etc). Enterprise-alike application typically use adapters into databases or CRM/ERP systems.
- **Search kernel and scalable architecture**. Large scale search engines are usually distributed onto several physical machines. The search kernel must be fit for this, and a framework for distribution, fault-tolerance and scaling are built on top of it.

- **Query frontend**. In front of the kernel and its distributed architecture, a query handling and result presentation frontend is placed. In its simplest form it is merely an HTML or XML template engine, but more sophisticated query preprocessing is often also included here.

Various methods have been developed to support efficient search and retrieval of text collections. Examples are *suffix arrays*[5][3], *inverted files* or *inverted indexes*[8][10], and *signature files*[2].

The performance of a search kernel is of critical importance when it comes to scaling of a search engine. The primary goal of FMS Search is to build a lean and flexible search engine kernel that is optimized to be part of very large scale search systems. This paper outlines the structures and algorithms of such a system along with a theoretical performance model. Furthermore, we provide empirical studies of the search kernel to verify the correctness of the performance model.

There are several other issues on designing a search kernel related to network handling, traffic, thread and process scheduling that are beside the discussions in this paper.

2 Preliminaries

The FMS Search Engine[7] searches a set of documents ,D. These documents are preprocessed and a catalog, C, is built. A catalog may contain different indexes, I. Each index may hold a different portion of each document (we might have a separate index for summaries, locations, body text, etc.). Each document may also have meta information (which may or may not be part in any index). This meta information will be denoted M_{doc}.

Searching is done by parsing and executing queries, Q. A query consists of a set of query terms, $\{q\}$. Query terms, q_i, can either be independent words, or several query terms may be grouped to a phrase query term. A phrase query term is a sequence of single terms, $q_1, q_2, \ldots, q_n$, that should be matched in the sequence given by the phrase term.

3 Overview

The overall FMS Search overall is shown in Figure 1.

A search engine consists of several domain mappings, enabling mapping from a query Q to a resultset R, i.e. :

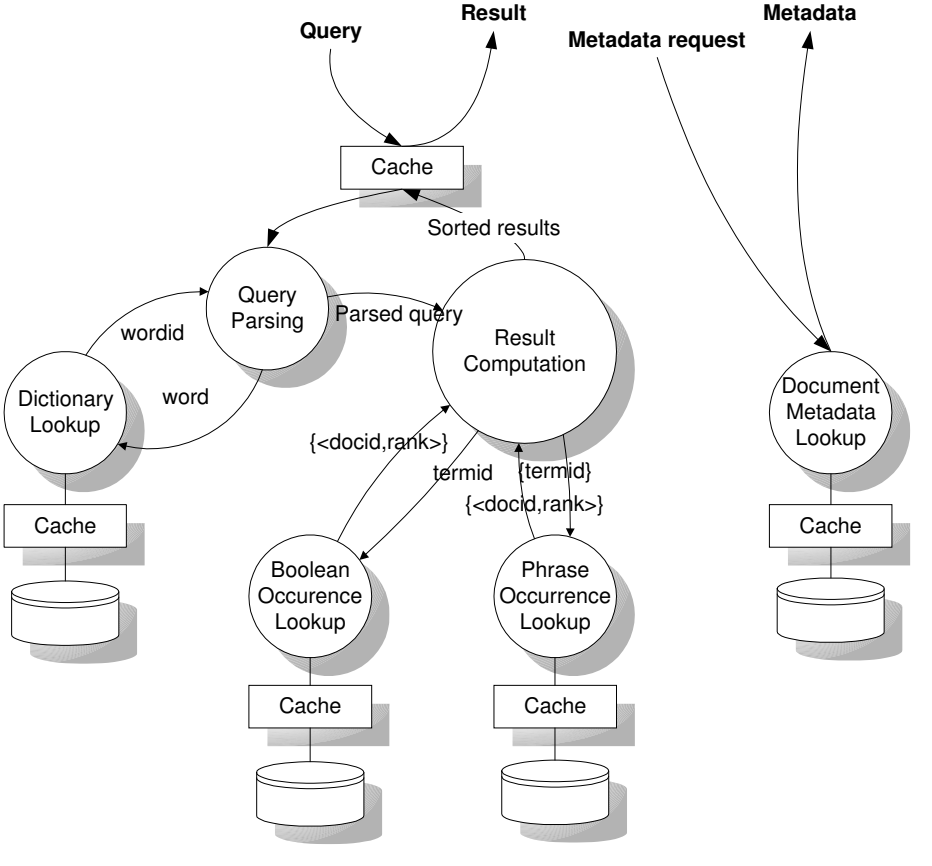


Figure 1. FMS Search overall anatomy

$$Q \rightarrow R \tag{1}$$

During this process, several substeps are performed by mapping between different domains. FMS Search currently uses the following domains:

- Query, Q
- Query term, q
- Term id, id_q
- Document id, id_{doc}
- Document metadata, M_{doc}

The mappings between the different domains will be described in more detail.

3.1 Query parsing and dictionary lookup

The first step of query processing is a combined parsing of the query and dictionary lookup of the query terms. Thus, we have a mapping from query to the set of query terms,

$$Q \rightarrow \{q\} \tag{2}$$

Also, by doing a dictionary lookup the query terms are mapped into their respective term ids by the mapping :

$$q \rightarrow id_q \tag{3}$$

A program, P, is also extracted based on the semantic found in the query, Q.

After this step of query processing the query consists of a set of term ids, $\{id_q\}$ and a program P describing how to compute the final result set.

3.2 Result Computation

After mapping query terms to query term ids, the next step is to do the real searching. That is, map from a term id to a set of documents in which the query term occur. From the set of documents for all the query terms, a final resultset, R, is computed based on the program P extracted from the query.

Depending on the nature of the query term, two different mappings can be performed in the index:

- **Single term mapping** If the query term is a singular term, then the mapping :

$$q \rightarrow \{id_{doc}\} \tag{4}$$

  will be performed. A reverse index occurrence file is used to perform the mapping. The structure of the file will be described in detail later in this paper.

- **Phrase term mapping** When the query term is a phrase, a different mapping must occur. The phrase is defined as a set of terms $\{q\}$. Thus, the mapping :

$$\{q\} \rightarrow \{id_{doc}\} \tag{5}$$

  gives the set of documents where the phrase $\{q\}$ are found. The structures and algorithms used for this mapping are described later.

3.3 Document information

A document D, that is indexed in the catalog may have some meta data that can be made available in the search engine. Examples of such metadata can be location (URL), size, date, title, etc.

Thus, FMS Search supports mapping from a given document id to the corresponding block of metadata for the document, $id_{doc} \rightarrow M_{doc}$.

4 Data structures

This section describes the data structures used in the search engine, as well as the algorithms used to perform the searching.

4.1 Dictionary

During query parsing, each query term, q, is mapped to its corresponding term id,id_q .

As shown in Figure 2, the dictionary is consists of records containing holding the word as a string, the accumulated number of occurrences of the word, and the accumulated number of documents for the term. The reason for holding the accumulated numbers instead of the separate numbers is

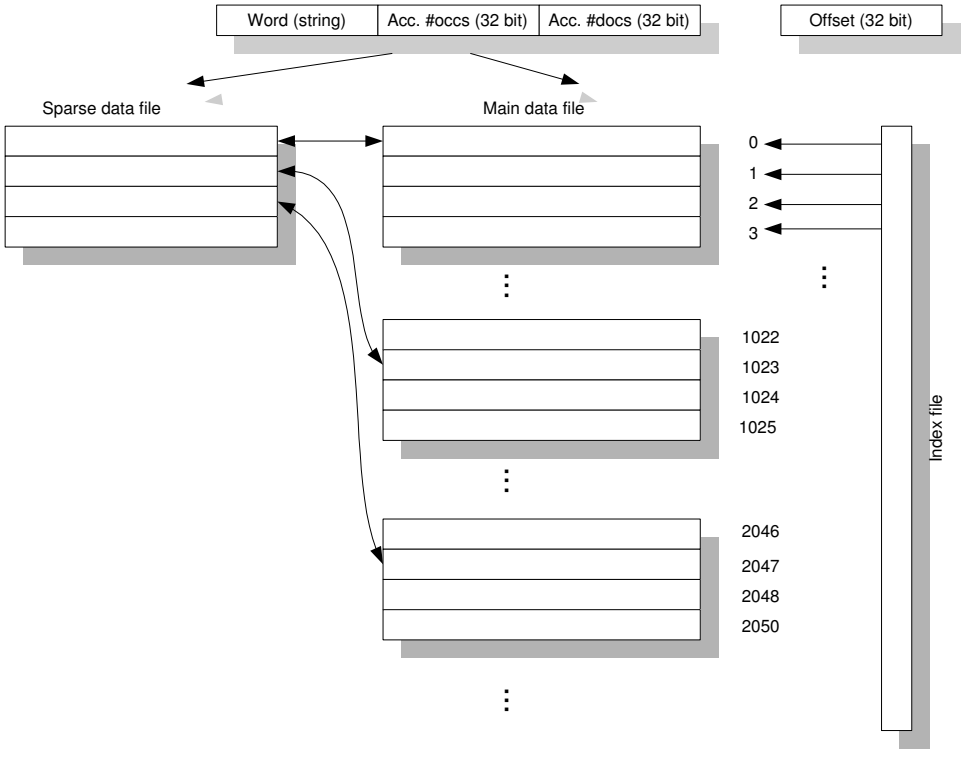


Figure 2. Dictionary structures

allow direct access in the term matching and the phrase matching phase of the search. This will come more apparent when we discuss the lookup algorithms later.

Since the length of the records are variable, a separate index file is used to get the position of the dictionary entry for a given word number. The word number is thus implicitly defined.

Using the index file and the data file for the dictionary, a simple binary searching algorithm can be used to lookup query terms. Letting $Dict$ denote the dictionary, $|Dict|$ denotes the number of records in the dictionary. Binary searching is a $O(log|Dict|)$ algorithm. Since the dictionary system is based on the data stored on disk files, the algorithm requires $O(log|Dict|)$ disk accesses.

The dictionary system of the FMS Search Engine kernel utilizes a binary search algorithm with two optimizations to reduce the number of disk accesses to $O(1)$ instead of $O(log|Dict|)$. A more detailed discussion around general optimization of binary search with variable access costs can be found in [6].

- **Sparse data**. By storing each 512th entry of the dictionary data and each corresponding dictionary index entry in main memory, and then ensuring that the initial delta value of the binary search algorithm is a power of 512, all comparisons required for the binary search will use the data available in main memory. This optimization requires $O(\frac{|Dict|}{512})$ main memory usage.

- **Buffering small deltas**.

  When using the sparse data as described above, the first phase of the binary search algorithm can determine two 512 entry areas of the dictionary where the record searched for may reside. Since the overhead of positioning the disk heads for a read operation is high relative to the time for a disk data transfer, it will pay off to buffer these 1023 entries when the delta gets below the value of 512.

The complete binary search algorithm can then be described with the pseudo-code shown in Figure 4.1.

4.2 Term matching

For matching a single term, it is required with a mapping from a term id to the set of documents containing the term. Also, some extra parameters may be required to perform ranking (i.e term frequency, term context, etc).

```
while (delta < |Dict|)
        delta ← delta << 1
delta ← delta >> 1
pos = delta − 1
while(delta > 0)
        if(pos&511) = 511)
                cmpres ← Compare(term, sparse[pos >> 9])
        elseif(delta == 512)
                ReadBuffer(pos − 512, pos + 512)
        if(cmpres = 1)
                pos ← pos + delta
        else
                pos ← pos − delta
        delta ← delta << 1
```

Figure 3. Binary search algorithm for dictionary lookup

The FMS Search Engine Kernel uses a format as shown in figure Figure 4 to store the required data. The fields of the records in the data file are described in Table 4.2.

The semantic interpretation of the *Context* field is independent of the search engine anatomy, and the interpretation during ranking is described in FMS Search Engine kernel configuration.

The entries are sorted in the file based on the document id. However, no term id entry is present in the record. This information is thus required to be determined before accessing this structure. As shown in Figure 4, this file contains the list of entries for each term id, sorted by document id within the list of each term.

Also, since the lists of document ids appear in the same order as the corresponding term ids (implicitly), the accumulated number of documents found in the dictionary structure can now be used to directly access the proper document id list in the term index. The length of the term list is easily determined by subtracting the accumulated number of documents from the next accumulated number (next entry in the dictionary file).

Since the format described in Table 4.2 does not contain a field describing the position within a document, we have no way of distinguishing between the different occurrences that may exist for a term in a document. Thus, we

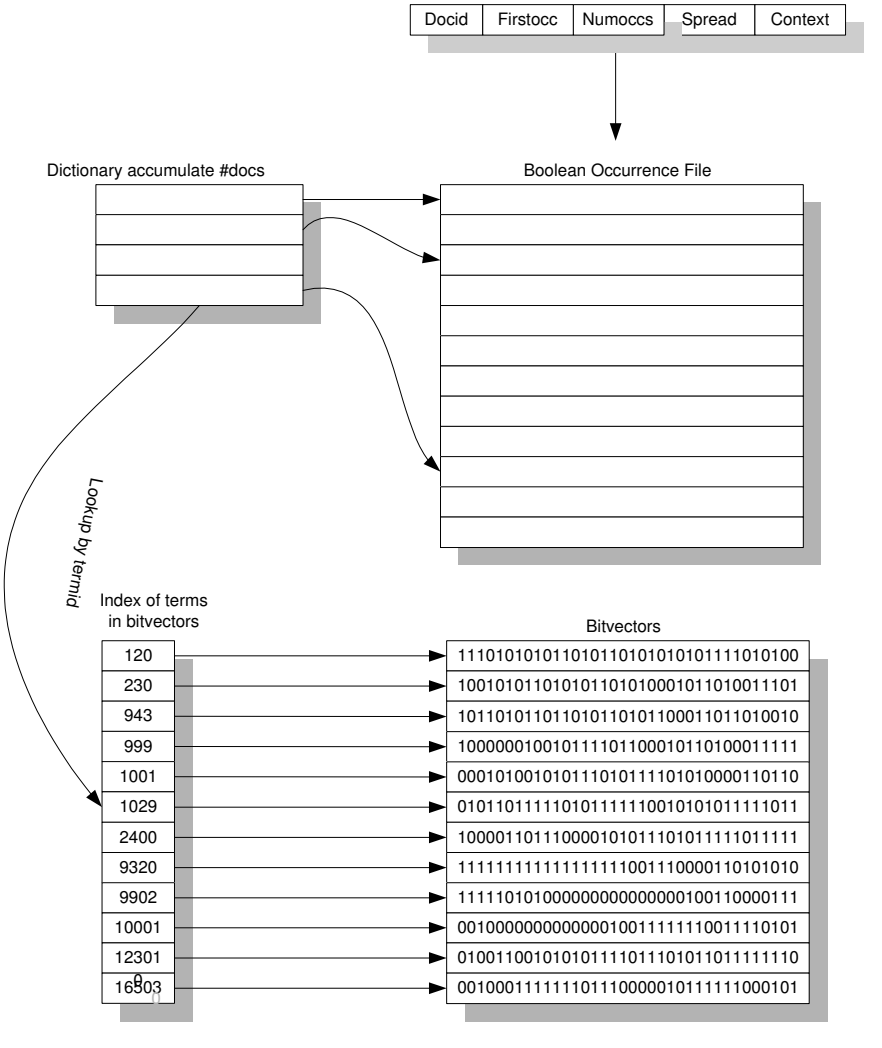


Figure 4. Boolean occurrence structures

Name	Sorting	Length (bits)	Description
Docid	Ascending	32	The id of the document
Firstocc	N/A	8	Position of the first occurrence of the term in the document (logarithmic scaled)
Numoccs	N/A	8	Number of occurrences of the term in the document (logarithmic scale)
Spread	N/A	5	A number indicating the spread of the term in the document
Context	N/A	3	3 bits that are used to describe the context of the term in the document (e.g. title, heading or body).

Table 1. Term index field descriptions

compress the occurrence information into the fields describing some statistics for the term within the given document.

4.3 Term bitmaps

Some terms tend to be very common in large document corpi. As shown in Table 4.2 each entry in the term file occupies 7 bytes. Furthermore, the very common terms should have less influence of the ranking than more seldom terms.

The FMS Search Engine kernel stores the document id lists of the more common terms in a different manner than the structure described in Section 4.2. Instead of storing a single entry per document, a vector of N bits is stored for each term, where bit i is set if the term is present in document i. This removes any context and ranking information found in the structure described in Section 4.2, thus we will have no way of distinguishing rank values of terms found in a bitvector. This might sound alarming, but the drilling mechanism described later solves most cases of this problem.

In Figure 4 the bitvector structure is shown along with the other structures used for terms.

The bitvector structure is also discussed in [11] with optimizations and generalization into bitslices.

Name	Sorting	Length (bits)	Description
Nextword	Ascending 1	32	The term id of the next consecutive word
Nextnextword	Ascending 2	32	The term id of second next consecutive word
Docid	Ascending 3	32	The id of the document
Position	Ascending 4	26	The position of the term within the document
First Context	N/A	2	Context information of first term
Second Context	N/A	2	Context information for the next term
Third Context	N/A	2	Context information for the second next term

Table 2. Phrase index field descriptions

4.4 Phrase matching

Recalling from Section 3.2 and in particular Equation 5, matching of phrases is matching of consecutive terms within a document.

The term matching format described in Section 4.2 does not contain entries for all the occurrences of a term within a document. Thus, being able to match phrases requires a more extensive index structure.

The FMS Search kernel employs a special version of Suffix Arrays [5]. In [9] and [1] the 'nextword' structure is introduced and discussed. The structure used in this paper is a generalization of this. An overview of the structures used is shown in Figure 5.

Let T be the text containing the terms t_0, t_1, :., t_N. Now, let T_i denote the term suffix t_i, t_{i+1},:.,t_N. Furthermore, let T_i^k denote the k-prefix of the term suffix T_i, t_{i+1}, t_{i+2},:.,t_{i+k}.

Now, let Pos be the lexicographically sorted array of all suffix of the text T. Pos is then the suffix array of T. Let Pos_k be the lexicographically sorted array of all k prefixes of T, T_i^k, that is $Pos[i] \leq_{lex} Pos[i+1]$ for all $i < N$, letting $\leq_{lex}$ denote an lexicographical ordering.

In FMS Search, we choose to use $k = 3$ in the phrase structure. The joint structure can then be summarized as in Table 4.4. The actual implementation uses multiple tables to avoid redundancy. This is illustrated in Figure 5.

Lookup of any given triple of terms, $[t_1, t_2, t_3]$, can now be done by a

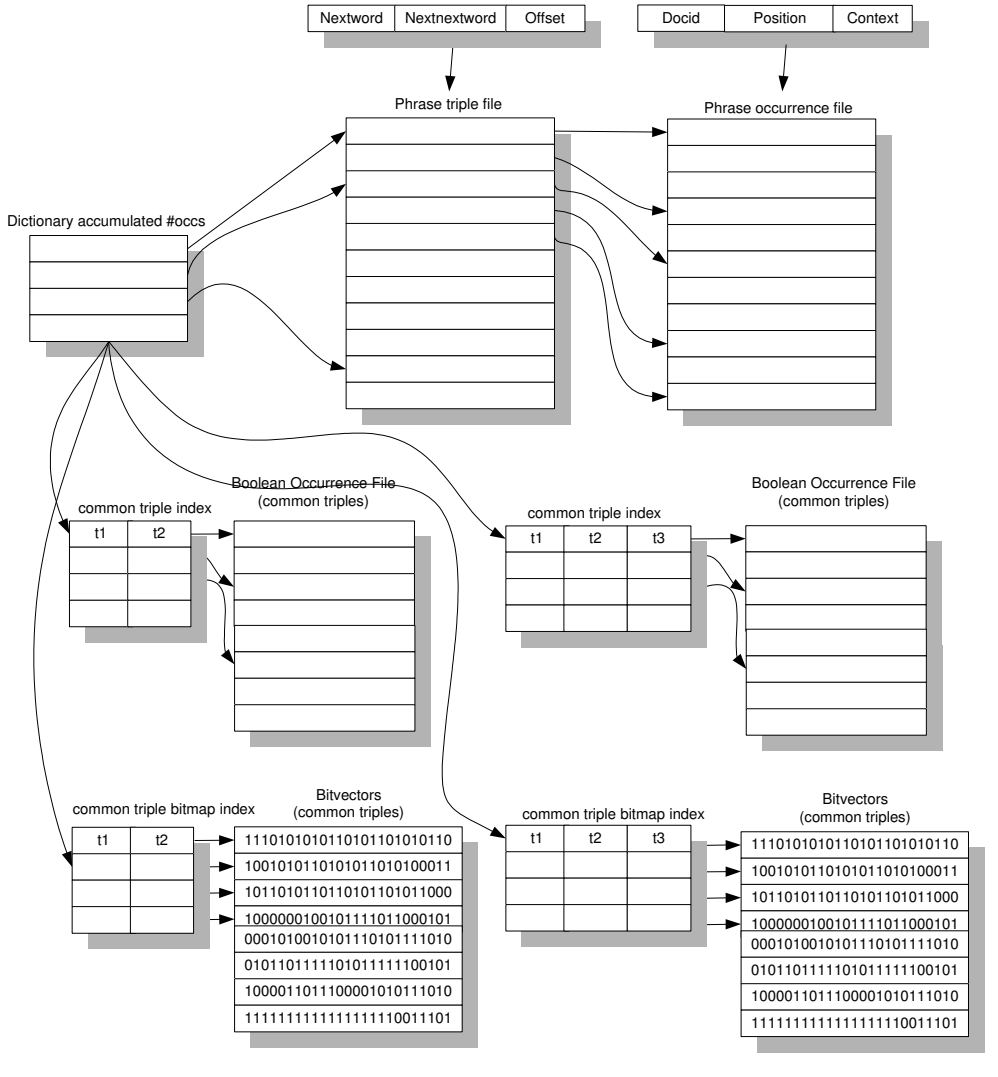


Figure 5. Phrase occurrence structure overview

```
m ← |Q|
if m = 2 then
  R ← Lookup([q1, q2])
elseif m = 3 then
  R ← Lookup([q1, q2, q3])
else
  start ← 1
  delta ← 3
  while start < m − 2 do
        R ← R ∩delta Lookup([qi, qi+1, qi+2]
        if start + delta > m − 2 then
              delta ← (m − 2) − start
        start ← start + delta
return R
```

Figure 6. Algorithm for merging phrases

binary search algorithm. In Figure 5 we observe that we have separate tables for triples and their occurrence data. Thus, by performing a binary search in the triple array, we can do a direct lookup into the occurrence table for the triple.

When performing a phrase matching operation of a pair of terms, $[t_1, t_2]$, the binary search in the triple array could result in a range, $[L, R]$. However, since $Pos[L] \leq_{lex} Pos[R]$ for all $L \geq R$, we can map directly into a larger portion of the occurrence array.

Performing phrase matching of phrases $[t_1, t_2, ..., t_m]$, where $m > 3$, requires multiple lookups to be performed. Letting $\cap_{delta}$ denote the intersection of phraseoccurrences with distance *delta*, the pseudo-code for performing the lookup of a phrase Q of any length is shown in Figure 4.4.

4.5 Performance

The lookup of a triple or a pair requires a binary search (algorithm will be almost identical to dictionary lookup). Thus, a $O(logn)$ algorithm can be used. As with the regular dictionary, we use sparse files to reduce the number of disk accesses in the binary search. (Those files are omitted from Figure 5 to enhance readability). Given the set of documents $\{D_i\}$ that totally adds up to the text T, it is obvious that the maximum number of

entries in *Pos* is $|T|$ (letting $|T|$ denote the number of terms in T). Thus, we have an upper bound for the number of triples. In most application, the number of unique triples is significantly less than $|T|$.

After performing a lookup of the triple (or pair), intersection of two sets R_1 and R_2 can be done in $O(R_1 + R_2))$ time. However, for most practical applications, the phrase occurrence table is stored on secondary storage, and needs to be loaded into primary memory before performing the intersection. Loading the data requires $\Theta(R)$ time, thus the intersection operations will take $\Theta(R_1 + R_2)$ time.

4.6 Common phrases

For very common triples, the number of occurrences can be quite large. To avoid performance problems for these triples (or pairs), we build some extra tables. By storing boolean occurrence tables (like the ones in Figure 4) for the common triples and pairs, we only need to store one entry per document the triple occurs in. Thus, this limits the memory requirement and disk load to $|D_i|$ for a triple or pair, instead of $|T|$, which is the case for the regular structure.

Of course, this structure cannot be used for searches of phrases with more than 3 terms.

4.7 Metadata lookup

Thus, FMS Search supports mapping from a given document id to the corresponding block of metadata for the document, $id_{doc} \rightarrow M_{doc}$.

The mapping $id_{doc} \rightarrow M_{doc}$ happens through a very simple two part datastructure:

- **Binary blob file**. Meta data for all documents in a big binary block file.
- **Memory mapped index**. A memory mapped array holding the offset for each id_doc into the blob file.

Thus, for a document i, we we need to read from the blob file from offset $M_{meta}[i]$ through $M_{meta}[i + 1]$. The interpretation of the blob object is left to the client software (for instance a web server).

5 Catalog, Indexes and Contexts

The FMS Search Engine Kernel can operate on different subsets of the entire dataset. Given a set of documents D_i accumulating to the text T. T consists of the the term $t_1, t_2, ..., t_N$. During indexing, each term has a context, $C(t_i)$. For HTML documents this can be the visual context (title, heading, etc). A partition of the data which shares the same set of possible contexts is usually grouped in a catalog. A catalog holds a separate dictionary, and separate boolean and phrase occurrence files. Multiple catalogs are often used to create searchable indexes of disjoint parts of the documents (e.g. the normal text as one catalog, and the address of the document as a different one).

Within each catalog, multiple indexes can be constructed. All indexes within a catalog shares the same dictionary, but has separate boolean and phrase occurrence files. Indexes can be any bitwise combination of the context. E.g. given a catalog **normal** having the contexts **title** and **body**. Then it is possible to construct an index **title** only holding terms that are found in the **title** context, and a different index **all** holding terms found in either contexts (**title** or **body**).

Multiple indexes with a common dictionary is implemented by repeating the accumulated fields for each entry in the dictionary for each index in the catalog.

Aside from the searchable content, indexing also creates two other classes of data:

- **Document metadata** is metadata about the document that are not necessarily searchable. The metadata is forwarded to the dispatching system, and is used for presentation of the search results.
- **Document attributes**. The attributes of a document can be multiple. Size, date, connectivity, and other qualitative measures can be used here. The document attributes are used for static ranking purposes.

The entity-relationship of the concepts used in the dataset is summarized in Figure 7.

6 Query processing

A query is interpreted using a stack machine in the FMS Search engine kernel. Thus, the query will be compiled down to a stack of query instructions

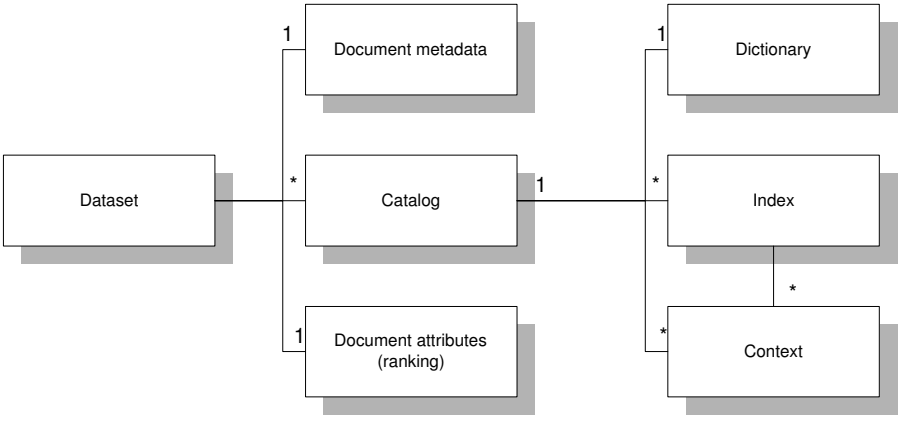


Figure 7. Entity-relationship between dataset components

before executing it.

The basic elements of the stack are:

- Term
- Phrase
- Result
- Operator

During execution of the stack, the stack is converted into the final search result. Thus, letting S_i denote a general stack element, and R denoting a result stack item, the execution of a query can be denoted $E : S_i \rightarrow R$. In Table 6, the different operators are described along with their arity. Note that the non-operator stack elements also can act as operators, thus they have a default behavior.

7 Result handling

During processing of a query in the search pipeline, a search result set is constructed. Thus, efficient datastructures and algorithms for result handling is crucial for the overall performance of the search engine. This section describes the datastructure used for result handling, and the algorithm used to operate on it.

Operator	Arity	Stack operations	Description
OR	$n \geq 2$	-	POP elements, compute the union of the parameter elements, then PUSH the OR'ed results.
AND	$n \geq 2$	-	POP elements, compute the intersection of the result of the parameter elements, then PUSH the AND'ed results.
NOT	$n \geq 2$	-	POP elements. Compute the difference between the result of the first element and the remaining elements of the operator. Then PUSH the final result.
RANK	$n \geq 2$	-	Ranks the result of the first parameter with the results of the $2 \ldots n$ parameter elements.
TERM	1	-	Search for the term found in the element. PUSH the result. A term can be of the form *catalog.index:term* or just *term*.
PHRASE	≥ 2	-	Search for the phrase consisting of the parameter terms and PUSH the result.
RESULT	1	Nil	A result set. Execution leaves the stack untouched.

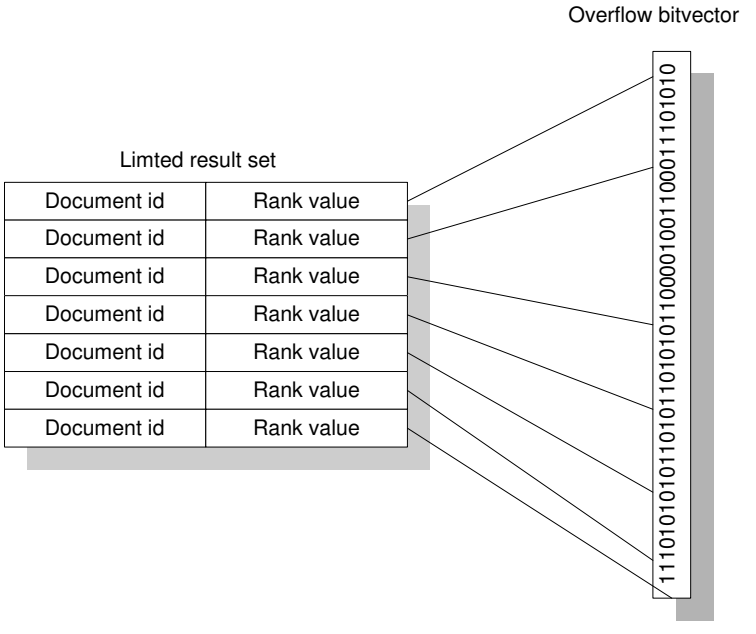


Figure 8. Resultset structure overview

7.1 Datastructure for results

Storing the results requires a datastructure serving three purposes:

1. **Performance**. Operations on the result set datastructure must be very efficient. For a high-performance search system hundreds of resultsets can be computed per second.

2. **Size** The size of the result set should be as small as possible. Result sets reside in main memory, which is a limited resource.

3. **Quality** The resultset datastructure needs to hold enough information to provide high quality results.

For each query, Q, the resultset, R, should be a set of tuples $< id_{doc}, rank_{doc} >$ holding documents ids and rank values.

Since the size of a result set can grow very large, a mechanism must be used to limit the space requirement for the structure. The ranking information is only used to sort the results based on their rank order. In most cases, there is no need to have the whole resultset sorted, since the user only observes the difference in ranking among the best results.

Thus, a simple solution would be only hold the t best results. This will however require holding the entire resultset during computation to ensure correct results.

Instead, we use a combined structure. An array of $< id_{doc}, rank_{doc} >$ tuples are combined with a bitvector with one bit per document in the dataset D_i.

Now, assuming that the query terms can be broken down in an array $[t_1, t_2, ..t_n]$, where $t_i <_p t_{i+1}$, letting $<_p$ denote an ordering based the expected average rank value, $E[r, t]$.

Thus, we have an ordering ensuring that we do the terms that are most likely to give the highest rank values first.

Letting l denote the maximum length to use for the array part of the resultset, and letting $|R(t_i)|$ denote the size of the intermediate results for term t_i, we add each term to the result set with the following check:

if $|R|+|R(t_i)| < l$ **then**
    $AddResultToArray(R(t_i))$
$AddResultToBitvector(R(t_i))$

7.2 Ranking

Given a query, Q, the result set R should be sorted according to the rank value. Each document in the resultset has a rank value for the query. The rank value have two main components, namely the static rank value and the dynamic rank value. The static rank value is a value computed for the given document D_i independently of the query, while the static rank is the rank for the document D_i with regards to a query Q. Each term in the query contribute to the dynamic rank value. Since we have the option to have boolean expression in the queries, there are two different types of terms in the query:

- **Positive terms**. Terms that are either part of an **OR** or an **AND** expression. These terms will have a positive effect on the ranking.
- **Negative terms**. Terms that are part of a **NOT** expression. These terms should have no impact on the ranking.

Now, defining the following function:

$$w(Q_i) = \begin{cases} 1 & \text{if } Q_i \text{ is a positive term.} \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

Then, we can define the rank value for a document D_i and a query $Q = Q_1, Q_2, \ldots, Q_n$ as :

$$r(Q, D_i) = r_S(D_i) + \sum_{j=0}^{j=n} w(Q_j) r_D(Q_j, D_i) \tag{7}$$

Where $r_D(Q_j, D_i)$ is the dynamic rank value for the document D_i for the query term Q_j. The dynamic rank value for a document D_i and a query term Q_j when Q_j is a single word term is defined as :

$$r_D(Q_j, D_i) = \frac{TFID[f_{Qj,d}] + FO[p_{Qj,Di,0}] + CB[c_{Qj,Di}]}{DT[log_2 f_Q j]}$$

where $f_Q j, Di$ denotes the term frequency of of term Q_j within the document D_i, $p_{Qj,Di,0}$ denotes the position offset of the first occurrence of term Q_j within the document D_i. $c_{Qj,Di}$ denotes the context(s) of the term Q_j in document D_i, and $f_Q j$ is the global frequency of the term Qj.

When, the query term is a phrase term, the dynamic rank value is defined as:

$$r_D(Q_j, D_i) = \frac{B_{phrase} + FO[p_{Qj,Di,0}] + CB[c_{Qj,Di}]}{DT[log_2 f_Q j]}$$

B_{phrase} denotes a possible rank boost value for phrases.

$TFIB$, FO, CB and DT are lookup tables to approximate the functions used for ranking based on the different ranking factors.

After the final resultset R has been computed, a bin sorting algorithm is applied over the array to produce the correctly ranked resultset.

7.3 Drilling

Recalling from Section 7.1, we have two parts in the result set datastructure, the array and the bitvector. Obviously, the bitvector does not provide any ranking information whatsoever. From the algorithm in Section 7.1, it is obvious that the array will not hold all hits in the result set unless l (the limit of the array size) is sufficiently large.

We also observe that the algorithm for adding term results to the final result work on entire subresults for terms. Thus, the array size can be significantly lower than the limit l. Which results to put into the array and which to put into the bitvector is now chosen based on the frequency of the term, which is a good estimate on the expected average rank value for that term, referring to 8.

By observing that within a **catalog**, there may be multiple indexes that overlap eachother, as illustrated in Figure 9, we can improve the fillgrade of the array, by performing a *drilling* operation. From Figure 9, we observe that in Catalog B, $I2 \subset I1$ and $I3 \subset I1$. A typical example can be when searching HTML documents. Then $I1$ could be the entire document, while $I2$ could be the title of the document (which is more important), and $I3$ could be the headings in the document (which also are more important than the general body). Then it should make sense to first try to fit all hits from the largest index into the array, and if not, go to the second largest, and so on. The assumption being placed here is that the ranking information retrieved from the sub-indexes are a close enough approximation to the ranking for the entire document, and thus we are able to get a much higher fillgrade of the array, and thus more hits with ranking information for the particular query.

By identifying which indexes can be used for this drilling algorithm, a set of links should be made to determine the sequence of indexes that are searched. For the example above, it would be : $I1 \rightarrow I3 \rightarrow I2$.

Then the result computation algorithm in Section 7.1 could be refined to try multiple indexes. The new proposed algorithm is shown in Figure 7.3.

8 FMS Search performance model

In order to understand performance, bottlenecks and trade-offs of the FMS Search system, a simple performance model will be useful. Recapping from earlier in this paper, the following mappings take place in order to move from a query, Q to a result set, R.

1. $Q \rightarrow \{q\}$, from a query to a set of query terms.
2. $q \rightarrow id_q$, from a query term to a term id.
3. $q \rightarrow \{id_{doc}\}$, from a term id to a set of document ids.
4. $\{q\} \rightarrow \{id_{doc}\}$, from a set of term ids (phrase) to a set of document ids.

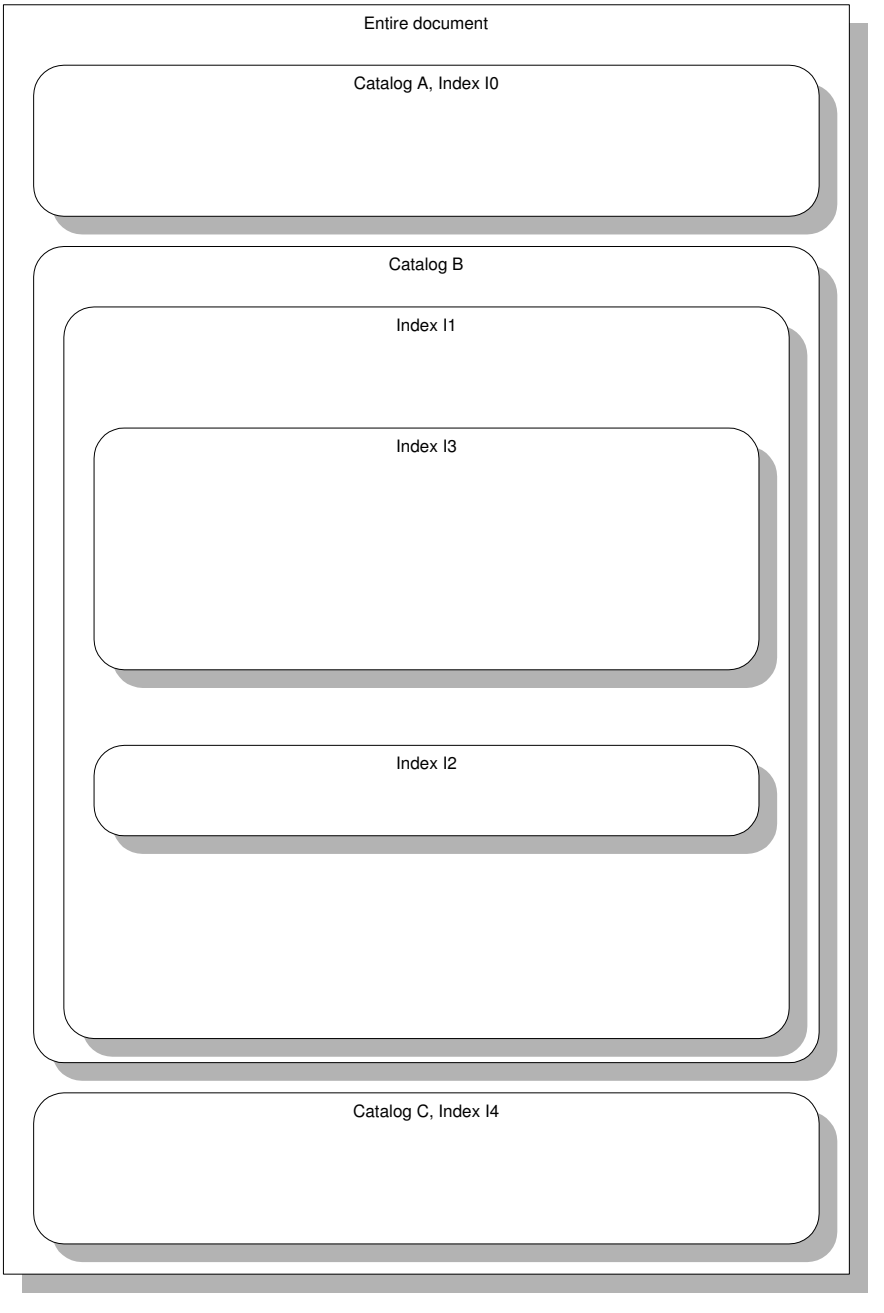


Figure 9. Catalog and Index set map

$I_{orig} \leftarrow I$
while $(Drill(I) \neq NIL)$ **and**$(\ |R| + |R\ I| > l$ **do**
    $I \leftarrow Drill(I)$
if$I \neq I_{orig}$**then**
    $R \leftarrow R + Bitvector(I_{orig}) + Array(I)$
else
    $R \leftarrow R + Array(I)$
$return R$

Figure 10. Result computation with index drilling

.

In these models we will make the assumption that the relation between number of hits for a given query and the number of documents searched is following Heaps law [4]:

$$R = Kn^{\beta} \tag{8}$$

Where K and β are constants between 0 and 1.

8.1 Query Parsing and Dictionary Lookup

The first mapping happens through query parsing. Query parsing is not thoroughly described in this paper, but there are no recursive constructs in the grammar, so the parsing is done in linear space and time to the query itself, $O(|Q|)$.

In order to perform the second mapping, which must take place for every term in the query, a dictionary lookup is required. The algorithm in Figure 4.1, is a simple binary search, thus it takes $O(log|Dict|)$ to run. The algorithm is guaranteed to have $O(1)$ access to the disk storage, given that we use $O(\frac{|Dict|}{512}$ memory space. A disk read will be for 1024 records of variable size. Letting $\overline{q}$ denote the average wordlength, the disk operations will be 1 seek and then $1024 \cdot (\overline{q} + |ptr|)$ bytes to read, where $|ptr|$ denotes the size of the pointer in the dictionary. Summing up:

Processing	$O(log\|Dict\|)$
Memory usage	$O(1)$
Disk transaction	1
Bytes read	$1024 \cdot (\overline{q} + \|ptr\|)$

8.2 Boolean occurrence lookup

From the dictionary lookup, the offset and length of a boolean occurrence list is implicit. Thus, this operation is a disk read operation only. Letting $hits(q)$ denote the number of hits for the query term q, we need to read $hits(q)$ bytes from disk in this operation. However, if $hits(q) > \frac{N}{8}$ there is only a bitvector that will be read for the first round, but in addition, drilling might take place and result in the read of a boolean occurrence list in addition.

Processing	$O(1)$
Memory usage	$\Theta(hits(q)) + \frac{N}{8}$
Disk transaction	2
Bytes read	$hits(q) \cdot 7 + \frac{N}{8}$

8.3 Phrase occurrence lookup

The phrase matching part has a more complex operational model. As described in Section 4.4, there is a binary search applied to get subresults for tuples and triples. Letting $|Q|$ denote the length of the query (number of terms), we will need to perform $\lceil |Q| \rceil$ binary searches towards a disk structure of size $O(N)$. The binary search is using a sparse index as described earlier, resulting in $\Theta(1)$ disk reads.

Letting $hits(q_i \dots q_j)$ denote the number of occurrences of the the n-tuple $q_i \dots q_j$, we will need to merge the retrieved subresults of the n-tuple with the current result set using a linear-scan merge. Thus, merging will have to take place $\lceil \frac{|Q|}{n} \rceil - 1$ times, where n is typically 3 with the datastructure used here.

Processing	$O(\|Q\|) \cdot max_i hits(q_i \dots q_j)$
Memory usage	$2 \cdot max_i hits(q_i \dots q_j)$
Disk transaction	$\frac{\|Q\|}{n} \cdot \Theta(1)$
Bytes read	$(\sum_{i=0}^{\|Q\|-n} hits(q_i \dots q_i + n)) \cdot 7$

8.4 Result Combination

For every query term, there will be a merging phase into building the main result set. Simplified, we are merging two results sets for every term in the query, and merging happens with a linear scan approach.

Given the use of result sets that have both an array with max l elements, and a bitvector - every merge step will take $\Theta(max(l, hits(q_i)) + \frac{N}{8})$, where $hits(q_i)$ denotes the number of hits for the term q_i of the query, and N

denotes the total number of documents being searched. So, for a query Q there will be $\Theta(\sum_{i=0}^{|Q|-1}(max(l, hits(q_i)) + \frac{N}{8}))$ which can be simplified into $O(|Q| \cdot (l + \frac{N}{8}))$.

8.5 Overall Performance Model

In order to utilize the performance model in scaling and planning of search engine installations, it is necessary to derive the model onto a few key parameters:

- **Corpus Size**. Number of documents in corpus of the engine.
- **Query load characteristics**. Average number of terms in query, $|Q|_{ave}$, average number of hits in result set $hits_{ave}$, proportion of query that is phrases (in average), p_{phrase}.

From [4] we derive that the size of a dictionary is related to the size of the corpus in a sub-linear model, but due to having a simplistic tokenization, the dictionary is for all practical purposes linear in growth to the corpus. So thus we assume:

$$|Dict| \sim N_{docs} \tag{9}$$

Furthermore, assuming that the number of results returned for a query is bound to the corpus by means of Heaps law we can state that $hits(Q) = O(\alpha \cdot N_{docs})$ where N_{docs} denote the corpus size, and α is a scaling factor dependent of the query mix.

Since there will be dictionary lookups for all query terms, there will always be $O(|Q|_{ave})$ disk transactions and using 9 we will have $O(|Q|_{ave} \cdot log(\alpha \cdot N_{docs}))$ processing.

For boolean occurrence lookups, using 8 we derive that there is $\emptyset(\alpha \cdot N_{docs})$ memory usage.

On phrase occurrence lookups, we can simplify the expressions to being $O(|Q|_{ave} \cdot \alpha \cdot N_{docs})$ processing, $O(\alpha \cdot N_{docs})$ memory usage, and $|Q|_{ave}$ disk transactions.

Letting the query having $p_{phrase} \cdot |Q|_{ave}$ phrase elements, and $(1 - p_{phrase}) \cdot |Q|_{ave}$ non-phrase elements, we can express the overall performance model as shown in Table 8.5.

Processing	$O(\|Q\|_{ave}) \cdot log(\alpha N_{docs}) + p_{phrase} \cdot \|Q\|_{ave} \cdot \alpha N_{docs} + l \cdot log(l)$
Memory usage	$O(\alpha N_{docs})$
Disk transaction	$O(\|Q\|_{ave}) + O(p_{phrase}) + O((1 - p_{phrase}) \cdot \|Q\|_{ave})$

9 Performance evaluation

Several experiments have been conducted in order to determine the performance curves of the FMS Search engine. The main goals is to determine the *service level* for different document volumes, query loads and query sets.

Service Level includes the following metrics and constraints:

- Query capacity - number of queries per second the system can handle.
- Average responsetime - The average time from a query is submitted till the response returned.
- Responsetime percentiles - Percentage of queries that return results within a given timeframe.

For the benchmarks conducted in this study, we measure capacity and performance by having at least 99.5% of the queries answered within 3 seconds, and with average response time lower than 0.4 seconds. Thus:

$$p_{99,5} = 3s \quad (10)$$

$$t_{ave} = 0.4s \quad (11)$$

Furthermore, we also measure CPU load and disk load to indicate that the performance model outlined in this paper is approximating the real behavior.

9.1 Benchmark configuration

We used a machine configuration identical to the one being used in the FAST/Overture production environment, as follows:

CPU	Dual 2.8Ghz Pentium Xeon
Memory	1GB Main memory
Disk	12 x 18Gb (10K rpm)
Disk bus	2xUW-SCSI channels

Documents	qps	disk kb	disk tps	CPU idle
3M	900	25	2000	45
4M	557	34	1720	41
5M	434	40	1560	42
6M	352	50	1350	43
7M	365	58	1450	35
8M	311	66	1315	37
9M	292	73	1310	35

Furthermore, we use a large sample query log from the "www.alltheweb.com" search destination site. The query log has 500K entries, randomly sampled from the live query log. Characteristics of the query log are :

Number of queries	500,000
Unique queries	110,245
Average # terms in query	2.643

All the queries performed in these studies have been conducted where the default boolean operator between all terms was **AND**.

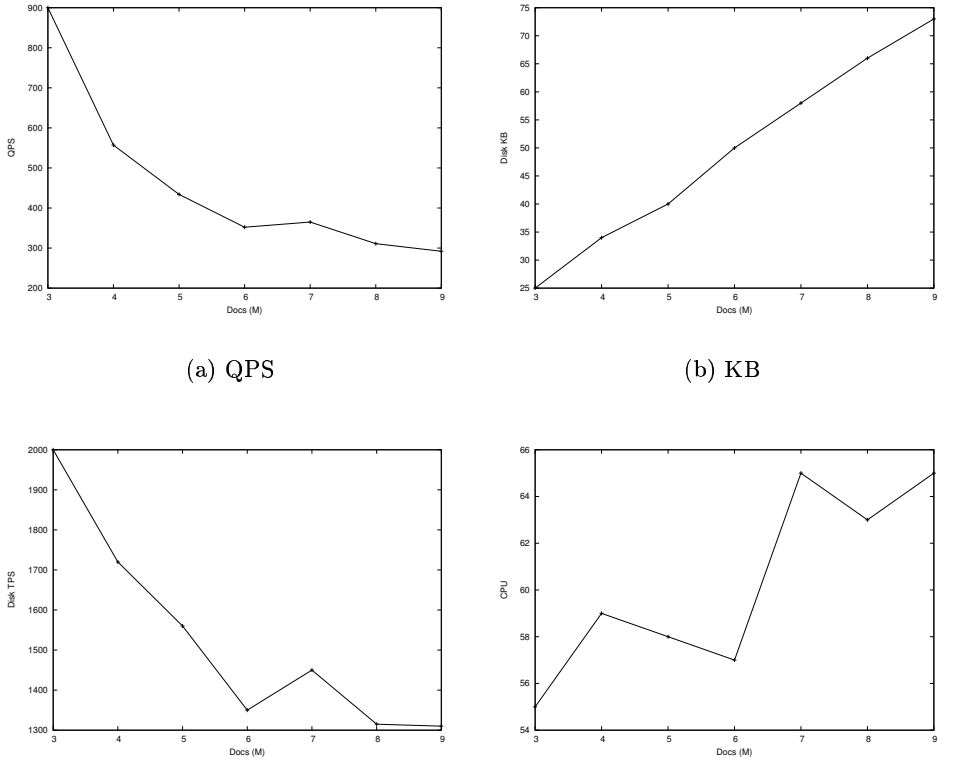
9.2 Performance for different data volumes

First, we indent to show the performance characteristic of the search kernel with different number of documents indexed.

From the curves we observe a super-linear relationship between the document size and the actual capacity. From the equations drawn above, one would expect performance to be constrained by resource capacities (CPU, disk, memory) and this is clearly observed in Figure 11. We see that transaction size and transaction rate has a -1 correlation which is expected, thus the amount of data streamed from disk is close to constant. Furthermore, we see a jagged, but still linear curve of CPU usage. Since we are relying on SCSI drives with command tagging, the CPU load from disk operations is fairly low.

We observe that the capacity is actually not linearly dropping of as we increase the number of documents on the node. This is mainly due to the mechanisms of drilling described in this paper. Thus, we limit processing and memory usage by using bitvectors for parts of the result set we expect to be large.

Observing the $KB \cdot TPS$ relation to be fairly constant in the document range, we draw the conclusion that the sweet spot of document size should



(a) QPS

(b) KB

(c) TPS

(d) CPU

Figure 11. Performance metrics for different document sizes

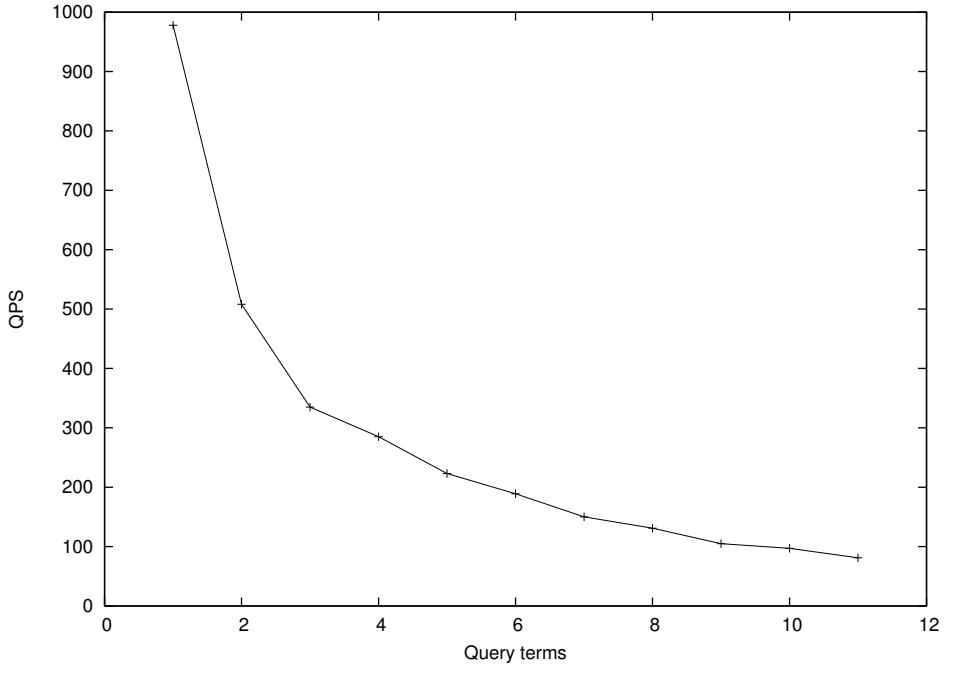


Figure 12. QPS as a function of number of terms

be expected to lie in the higher range of our test range.

9.3 Performance for different query complexity

The sample in 9.1 were now used to generate test sets with queries containing 1, 2, ..., 12 terms. The test was run applying a rate of queries as high as possible and still maintaining the SLA as described in 10. The rate (as queries per second) indicates the performance for the given subset of queries. The rates are shown in Figure 12.

9.4 Performance for different #hits

To investigate the performance of the search engine for various ranges of hits, we create a test query set as follows:

- **Lower bound sets.** Create subset for all queries with less than 1, 10, 100, ... hits.

Structure	Cache size (entries)	Hit ratio
Dictionary	100,000	84 %
Boolocc	200,000	64 %
Phraseocc	1,000.000	43 %
Bitvector	20,000	77 %
Docsum	10,000	53 %

- **Upper bound sets.** Create subsets for all queries more more than 1, 10, 100, ... hits.

Then, each of these querysets were used for performance testing, and the curves are shown in Figure 13 and Figure 14. The curves are logarithmically scaled. We observe the effect of drilling when the hitcounts are high (lower boundaries), causing the scaling behavior to be very attractive. Also the IO load support that. The Figure 14 shows that the drilling break-point is around 1M hits.

9.5 Cache hit-ratios

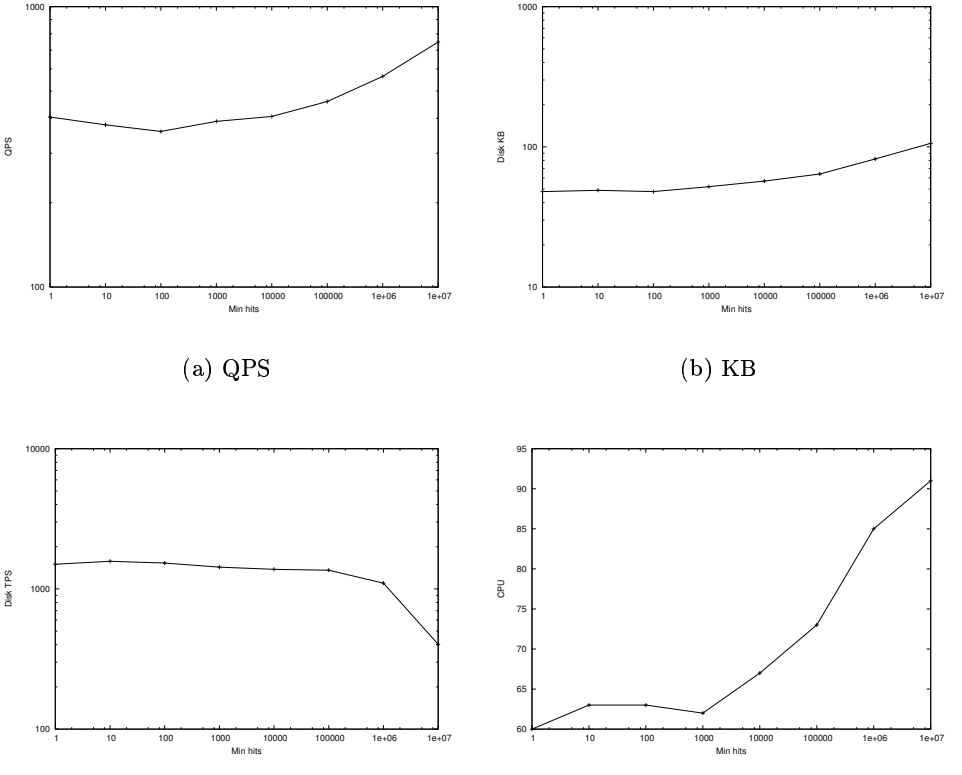
Running the combined query set, we also measured the cache hit ratio of the caches for the different structures with corresponding sizes of the cache. The results are shown in Table 9.5. In the scope of this paper we did not experiment with the cache sizes, since it would only make sense to do this in the setting of larger system with caching at levels outside of the search node as well.

10 Conclusive remarks

We have described the outline and main concepts of the FMS Search Engine kernel. The combination of inverted files, suffix structures, drilling and bitvectors we are able to build a search kernel with very good performance characteristics. The theoretical model we derive suggest performance to be linear in query terms and sub-linear in number of documents.

Empirical studies reveal a wide sweet-spot for a reasonable node configuration and performance characteristics that are in line with the theoretical model.

The FMS Search kernel is an industrial search kernel that have been proven to have performance and reliability for a wide variety of applications. Since the design of the kernel that was described in this paper, there has



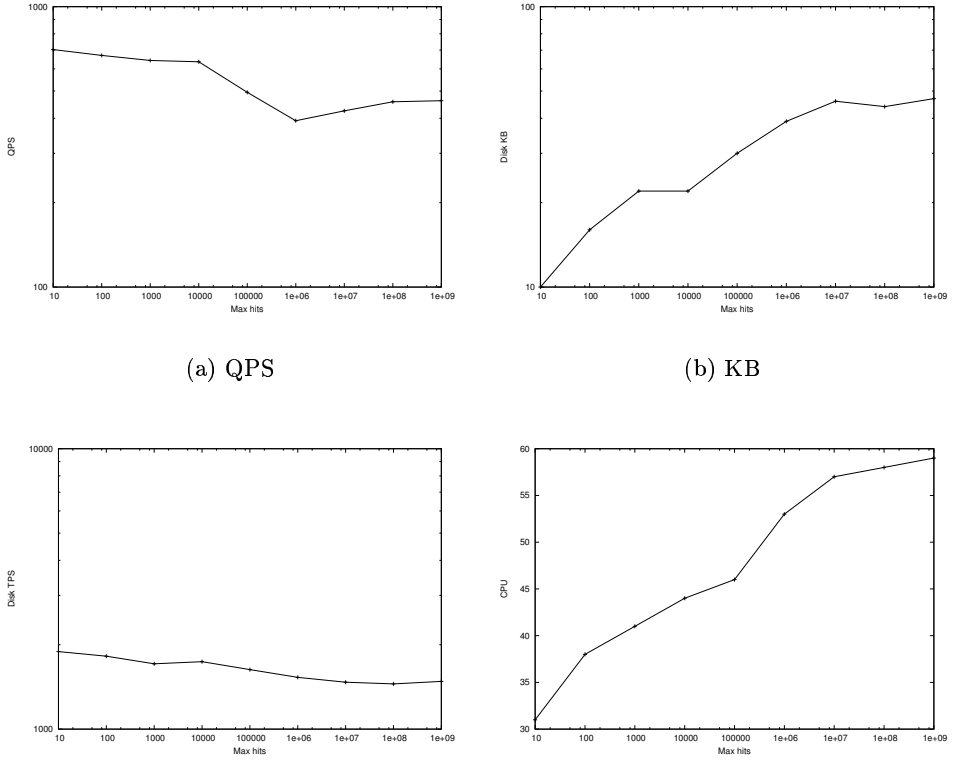
(a) QPS

(b) KB

(c) TPS

(d) CPU

Figure 13. Cumulative performance metrics for lower bound of numhits



(a) QPS

(b) KB

(c) TPS

(d) CPU

Figure 14. Cumulative performance metrics for upper bound of numhits

been an enormous amount of development of the kernel, adding support for compressed datafiles, proximity ranking operators, parallel evaluation of query terms, and so on. Still the basic concepts described in this paper are valid and key to the search kernel design and performance.

References

[1] BAHLE, D., WILLIAMS, H. E., AND ZOBEL, J. Efficient phrase querying with an auxiliary index. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval* (2002), ACM Press, pp. 215–221.

[2] FALOUTSOS, C., AND CHRISTODOULAKIS, S. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Office Information Systems 2*, 4 (October 1984), 267–288.

[3] GROSSI, R., AND VITTER, J. S. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). pp. 397–406.

[4] HEAPS, H. S. *Information Retrieval: Computational and Theoretical Aspects.* Academic Press, 1978.

[5] MANBER, U., AND MYERS, G. Suffix arrays: a new method for on-line string searches. *SIAM Journal of Computing 22*, 5 (Oct. 1993), 935–948.

[6] NAVARRO, G., BAEZA-YATES, R. A., BARBOSA, E. F., ZIVIANI, N., AND CUNTO, W. Binary searching with nonuniform costs and its application to text retrieval. *Algorithmica 27*, 2 (2000), 145–169.

[7] RISVIK, K. M. A search processor and method for retrieval of data and the usage in a search engine. International Patent PCT/NO99/00233, 1999.

[8] SALTON, G. *Automatic Text Processing: The Transformational, Analysis, and Retrieval of Information by Computer.* Addison-Wesley, Reading, Massachusetts, U.S.A., 1989.

[9] WILLIAMS, H. E., ZOBEL, J., AND ANDERSON, P. What's next? - index structures for efficient phrase querying. In *Proceedings of the Tenth Australasian Database Conference* (1999).

[10] WITTEN, I. H., MOFFAT, A., AND BELL, T. C. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[11] ZOBEL, J., MOFFAT, A., AND RAMAMOHANARAO, K. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst. 23*, 4 (1998), 453–490.

The FAST Distributed Processing Architecture (DPA) and its Application for a Large-Scale Search Engine

Knut Magne Risvik
Overture Services AS
P.O. Box 4452 Hospitalsløkkan
NO-7418 Trondheim, Norway
knut.risvik@overture.com *

Børge Svingen
Trondheim, Norway
bsvingen@borkdal.com

Tor Egge
Overture Services AS
P.O. Box 4452 Hospitalsløkkan
NO-7418 Trondheim, Norway
tor.egge@overture.com

Arne Halaas
Department of Computer Science
Norwegian University of Science and Technology
Trondheim, Norway
halaas@idi.ntnu.no

Abstract

We design a general framework for shared-nothing parallel computing. The framework is designed with an intention to be scalable of processing capacity and data volume capacity. Furthermore, a performance model for the architecture is derived, along with an algorithm for construction of scalable clusters. The DPA system is deployed for a large-scale web search engine, namely the FAST Web Search system. We design the distribution and scheduling functions and evaluate the performance of multiple configurations. The experiments show that the DPA system possesses the given scaling parameters and the latency is in concordance with the derived analytical model.

*Primary contact for this paper

1 Introduction

A large class of problems can be described in such a way that the data to be operated on can be distributed on a set of nodes, the problems can then be partially solved on each node, and the results are then accumulated.

This paper describes an architecture for such a system, the FAST Distributed Processing Architecture, and proves that the architecture is linearly scalable both with respect to the size of the data volume to be processed, and with respect to the number of problems to be solved.

In [7] we are introduced to the terms *SISD*, *SIMD*, *MISD* and *MIMD* as way of describing parallel systems. This architecture outlined in this paper is clearly a *MIMD* based one.

We are focusing on how FAST DPA can be used as a scaling architecture for search engine. Parallel Information Retrieval and search engines in particular have been a subject for a lot of research.

The paper has two main parts. The first part outlines the framework for DPA as a general architecture for solving problems with certain characteristics. We derive expressions for scaling and performance properties in the general sense.

The second part of the paper uses the framework as an architecture for a scalable search engine. We use the findings in [13] to analyze the performance and scaling of a search engine based on the FAST search kernel and the FAST DPA framework for scaling.

2 Related Work

Distributed architectures for Information Retrieval has been introduced in many settings. [10] is an example of an early one.

A shared-nothing approach, quite similar to the one being described in this paper was presented in [15]. Harvest was a resource discovery and information retrieval system that also had a distributed approached [4, 3]. The NOW (Network of Workstations) was the foundation of the Inktomi Search Engine, and has been presented and discussed in various settings [1, 8, 5].

Different approaches has also been outlined in [12] (for digital libraries), [14] (P2P based) and in [11].

3 Definitions

A data collection D is given. On this collection an equivalence relation $\sim$ is defined, and from this the equivalence classes form a partition $\mathcal{P} = \{D_i\}$. Due to this, $\forall D_i, D_j (D_i \in \mathcal{P}, D_j \in \mathcal{P} \Rightarrow D_i \cap D_j = \emptyset)$ and $\bigcup_{D_i \in \mathcal{P}} = D$.

Being an equivalence relation, $\sim$ fulfills the following requirements ([9]) :

- $(d, d) \in \sim forall d \in D$ (reflexiveness)
- $(d_1, d_2) \in \sim \Rightarrow (d_2, d_1) \in \sim$ (symmetry)
- $(d_1, d_2) \in \sim and (d_2, d_3) \in \sim \Rightarrow (d_1, d_3) \in \sim$ (transitivity)

On the subsets of D, a function $\sigma : \mathcal{P}(D) \rightarrow \mathbb{N}$ gives a measure of the actual data size.

A set of problems P is then given. Each $p \in P$ is of the form $p = \{P_i, \mathcal{P}_i, t\}$, where P_i is a problem instance, $\mathcal{P}_i$ is the subset of $\mathcal{P}$, such that $\bigcup \mathcal{P}_i \subset D$, that is of relevance to the problem, and t is the time at which the problem enters the system.

It is assumed that the set of problems P follow a Poisson distribution ([6]) characterized by the average λ, so that the probability of k problems arriving during a time unit is equal to

$$P(k) = \frac{e^{-\lambda} \lambda^k}{k!} \tag{1}$$

The number of problems arriving in non-overlapping intervals are therefore considered independent.

4 The Main Components of the FAST DPA

A central concept in the FAST DPA is the node. A node is an abstract processing unit; physically a node may consist of several workstations, or a single workstation may constitute several nodes. The nodes may be grouped into the following classes :

- **Processing nodes.**

  The set of nodes N_{proc} is used to solve the set of problems P – n is here equal to the number of equivalence classes given by $\sim$. A function $\phi : N_{proc} \rightarrow \mathcal{P}(\mathcal{P})$ specifies how the data set D is distributed to the set of nodes.

- **Problem distribution nodes.**

  A set of nodes N_{distr} distributes the problem $p = \{P_i, \mathcal{P}_i, t\}$ to the set of processing nodes used to process the problem. This set is given by the function $\delta : P \rightarrow \mathcal{P}(N_{proc})$, which is decided dynamically.

- **Result accumulation nodes.**

  Upon completion of the problem processing, the results are accumulated by the set of nodes N_{acc}.

- **Data preprocessing nodes.**

  In some cases the data D on which the problems will work need to be preprocessed. A set of nodes N_{pre} will serve this task.

An illustration of this architecture is given in Figure 1.

Thus, to solve a problem $p = \{P_i, \mathcal{P}_i, t\}$ the following steps are performed :

1. **Distribute**

   The problem $p = \{P_i, \mathcal{P}_i, t\}$ is distributed to the subset $\delta(p)$ of N_{proc} — δ is chosen so that $\bigcup_{N_{proc,i} \in \delta(p)} \phi(N_{proc,i}) \subseteq \mathcal{P}_i$. Thus, each processing node $N_{proc,j} \in \delta(p)$ will have an instance p_j of the problem p at the time $t + t_d$, where t_d is an expression for the latency of distributing the problem down to the problem solving node.

2. **Parallel solving**

   Each instance p_j will be solved in parallel on the processing nodes $\delta(p)$. Solving the instance p_j on the processing node $N_{proc,j}$ will take $t_{proc,p,j}$ time. Thus, the total solving time for all the problem instances p_j is

   $$t_p = \max_j \{t_{proc,p,j}\} \tag{2}$$

3. **Result accumulation and merging**

   Upon completion of the parallel solving process, the results from the processing nodes are accumulated and merged into the final result.

5 Architecture

This section will in detail describe the architecture of the FAST DPA, and, as part of this, describe how the functions ϕ and δ should be selected.

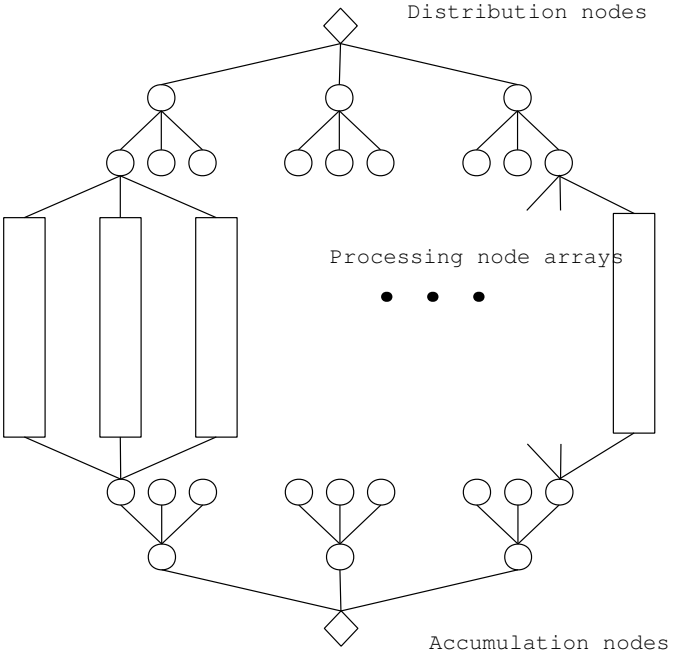


Figure 1. The FAST DPA

5.1 The Processing Node

Each processing node $N \in N_{proc}$ is assumed to have the following performance specifications :

- An average of k_{proc} problems can be handled in a time unit.
- Up to a data amount of σ_{max} can be handled. It is assumed that $\sim$ is decided so that $\max_{D_i \in \mathcal{P}} \sigma(D_i) \ll \sigma_{max}$.
- Problems take an average time t_{proc} to solve.

5.2 The Problem Distribution Node

Each problem distribution node $N \in N_{distr}$ is assumed to be able to distribute problems to up to k_{distr} other nodes.

5.3 The Result Accumulation Node

Each result accumulation node $N \in N_{acc}$ is assumed to be able to accumulate results from up to k_{acc} other nodes.

5.4 The Two-Dimensional Processing Node Array

As was described in the previous section, a single processing node can solve a certain number of problems on a certain amount of data. If $\sigma(D) > \sigma_{max}$ or $k_{proc} < \lambda$, then several data processing nodes must be used.

The data processing nodes are assumed to be arranged in an array, as shown in Figure 2. Here each column $N_{proc,j} = \{N_{proc,1,j}, N_{proc,2,j}, \ldots, N_{proc,r,j}\}$, for $j \in [1, c]$, contains a complete copy of the data D, while each node within the columns contains non-overlapping parts of D. The set N_c is defined by

$$N_c = \{N_{proc,j}, j \in [1, c]\} \tag{3}$$

The problem distribution now consists of two parts :

- First, each problem $p = \{P_i, \mathcal{P}_i, t\}$ is distributed to one of several replicas of the data — these replicas are in Figure 2 represented by columns.
- Second, the problem p is distributed to the nodes $\delta(p)$.

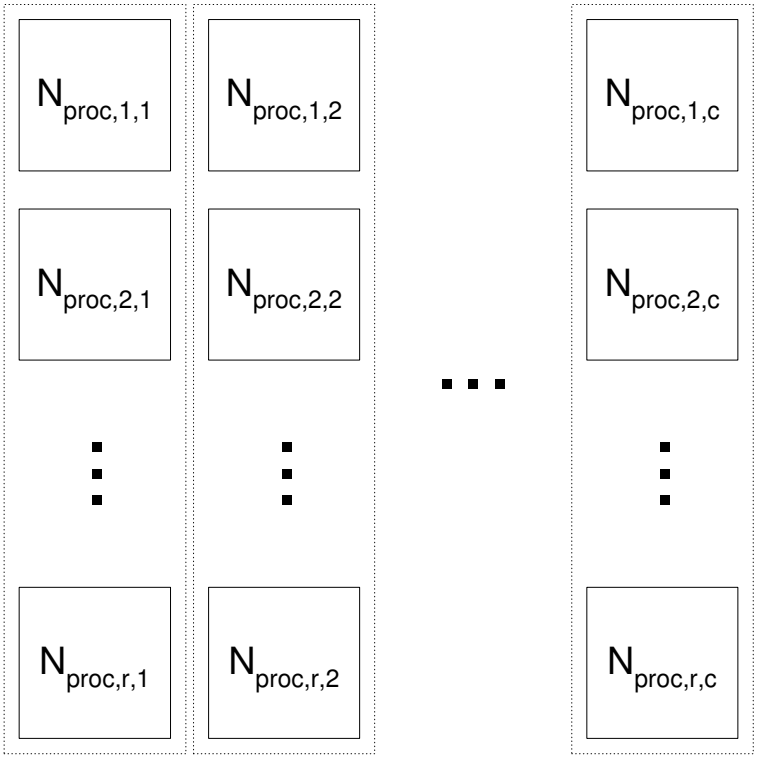


Figure 2. The Processing Node Array

In order to support the above points, the set of distribution nodes are divided into two parts, $N_{distr} = N_{distr_1} \cup \bigcup_{j \in [1,c]} N_{distr_2,j}$, where N_{distr_1} distributes each problems to one of the columns, while $N_{distr_2,j}$ distributes the problems within column j.

It is now possible to specify the functions $\phi : N_{proc} \rightarrow \mathcal{P}(\mathcal{P})$ and $\delta : P \rightarrow \mathcal{P}(N_{proc})$ in more detail. This is done is follows :

- The specification of ϕ is simplified by the fact that each column $N_{proc,j}$ contains identical data — a function $\phi_{\mathbb{N}} : \mathcal{P} \rightarrow \mathbb{N}$ is used to map each $D_i \in \mathcal{P}$ to one of the rows $i \in [1, r]$ in Figure 2, and ϕ can then be given by $\phi(N_{proc,i,j}) = \phi_{\mathbb{N}}^{-1}(i)$.

  The function $\phi_{\mathbb{N}}$ will be dealt with in Section 5.6.

- Since the distribution of problems take place in two separate operations, as stated above, the specification of the function $\delta : P \rightarrow \mathcal{P}(N_{proc})$ can be facilitated by using a function $\delta_1 : P \rightarrow N_c$ to specify the column $N_{proc,j}$ which should handle the problem. A function $\delta_2 : (P, N_c) \rightarrow \mathcal{P}(N_{proc,j})$ can now specify the actual nodes within that road. The function δ is thus given by

$$\delta(p) = \delta_2(p, \delta_1(p)) \tag{4}$$

  The definition of δ_1 and δ_2 is application dependent, and will be defined in description of application of this architecture.

5.5 Data Replication

If $k_{proc} > \lambda$, then a single processing node cannot keep up with the problems in P. It is therefore necessary to replicate the data, so that each of the replica work on a separate, non-overlapping, subset of D.

Problem distribution nodes are used to distribute each problem p to one of the columns — the decision of which column to use is made by the function δ_1, as described in the previous section.

To do the actual distribution of the problems, the nodes N_{distr_1} are used. N_{distr_1} are organized in the form of a graph, as shown in Figure 3 — the dotted parts of the figure are not considered part of the DPA, and will normally be provided in the form of a router. More specifically, N_{distr_1} is a forest of k_{distr}-ary trees — the reason for using a forest of trees instead of a single tree will be explained in Section 5.8.

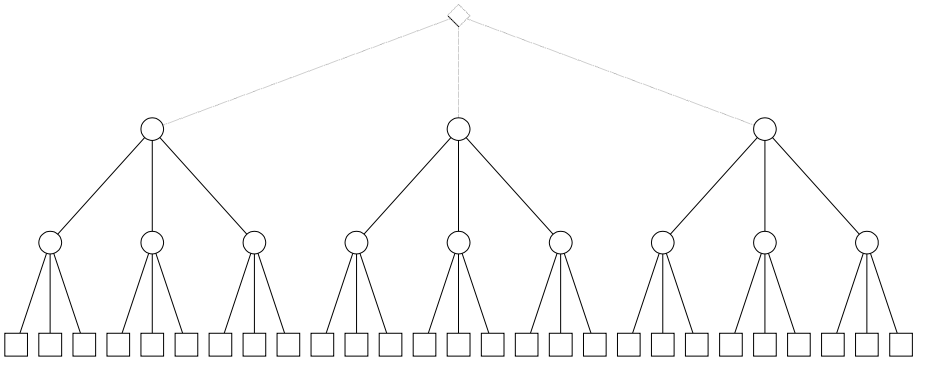


Figure 3. An Example Distribution Tree

5.6 Data Distribution

If $\sigma(D) > \sigma_{max}$, then the data D must be distributed to several nodes. The function $\phi_{\mathbb{N}} : \mathcal{P} \rightarrow \mathbb{N}$ specifies how this is done. A simple function $\phi_{\mathbb{N},uniform}$ to distribute the data D is to uniformly distribute the data volume equally on each processing node. More complicated functions can be utilized with domain knowledge of the application. It can thus be viewed as a packing problem, but this will not be dealt with here.

In order to distribute the problems to the set of nodes that contain data relevant to the problem, a tree similar to the ones in Figure 3 is used.

5.7 Result accumulation architecture

As each of the processing nodes finish solving parts of the problems, the results can be processed by the result accumulation nodes N_{acc}. The nodes in N_{acc} are organized as a tree, as shown in Figure 4, where the arity of the tree is given by

5.8 Fault Tolerance

There are basically two things that can go wrong given the architecture described in the previous sections :

- One of the nodes may stop functioning.
- One of the network connections may become broken.

In both of these cases, it is important that the system handles the situation as gracefully as possible. The following two points are used as guidelines

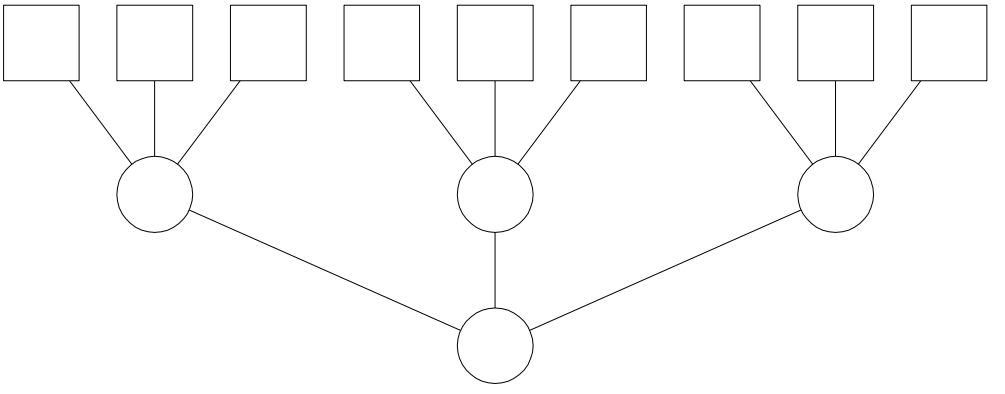


Figure 4. An Example Result Tree

in order to achieve this :

- It is not acceptable that some of the data in D is not available.
- It is acceptable that the performance is reduced until the error is corrected.

The FAST DPA fault tolerance strategy adheres to this by implementing the following points :

- If there is some kind of error inside one of the columns, then this column is marked as being faulty (or fawlty).
- If there is some kind of error in parts of the distribution of accumulation trees, then the top nodes of these trees are marked as faulty.
- If all the nodes that a distribution node distributes problems to are marked as faulty, then the distribution node is also marked as faulty.
- The distribution nodes will not distribute problems to faulty nodes.

The result of this is that when errors occur, the performance goes down, but it will continue to function correctly. It is, however, important that the top level problem distribution forest N_{distr_1}, and the result accumulation forest N_{accum_1} are in fact forests, and not a single tree, since in the latter case the whole system would go down if the root of the tree became faulty.

The number of trees in the forest is given by k_{distr_0}, in the figure $k_{distr_0} = 3$.

6 Performance

In previous sections, we found expressions for the total time of solving a problem $p = \{P_i, \mathcal{P}_i, t\}$. Now, we have defined that the problem distribution nodes are organized in a forest structure. The forest was divided into two different sub-forests, N_{distr_1} and $N_{distr_{2,j}}$ for all $j \in [1, c]$. The top level fanout was defined to be k_{distr_0}, while the fanout in the two sub-forests was defined as k_{distr_1} and k_{distr_2}, respectively.

Thus, the depth of the different sub-forests of distribution nodes can be expressed as:

$$\begin{aligned} depth(N_{distr_1}) &= \log_{k_{distr_1}} \frac{r}{k_{distr_0}} + 1 \\ depth(N_{distr_2}) &= \log_{k_{distr_2}} \frac{c}{r} \end{aligned} \tag{5}$$

Thus, the total depth of the distribution forest is:

$$\begin{aligned} depth(N_{distr}) &= depth(N_{distr_1}) + depth(N_{distr_2}) \\ &= \log_{k_{distr_1}} \frac{r}{k_{distr_0}} + \log_{k_{distr_2}} \frac{c}{r} + 1 \end{aligned} \tag{6}$$

Recalling that t_{lat} denotes the latency for a problem distribution node N_{distr} to distribute the problem $p = \{P_i, \mathcal{P}_i, t\}$ to the k_{distr} nodes below it, the total time for distributing a problem p to the subset $\delta(p)$ of processing nodes is:

$$t_d = depth(N_{distr}) t_{lat,distr} \tag{7}$$

The processing time for a problem instance p_j on the problem solving node $N_{proc,j}$ is denoted $t_{proc,j}$. For solving a problem p, the time spent by the processing nodes can thus be expressed as:

$$t_p = \max_{j \in \delta(p)} t_{proc,j} \tag{8}$$

The result accumulation nodes are also organized in a forest structure. We assume that the fanout of the accumulation node forest is always k_{accum}. Thus, the depth of that forest can be expressed as :

$$depth(N_{accum}) = \log_{k_{accum}}(rc) \tag{9}$$

The latency for a result accumulation node to accumulate the result from k_{accum} nodes and merge those into a new result is dependent of the incoming

result set size. Letting the function $\eta(\{R_j\})$ denote the latency for the result accumulation and merging process, where $\{R_j\}$ denotes the set of incoming result sets for the accumulation node $N_{acc,j}$.

Since merging of the results requires synchronization on each level in the accumulation result, we define a function $\zeta : \{N_{accum}\} \rightarrow \{N_{accum}\}$ specifies how the accumulation nodes are distributed into levels in the result structure. We can now define the total latency for accumulating the results down the entire tree of accumulation nodes:

$$\begin{aligned} t_a &= \sum_{k=0}^{depth(N_{accum})} max_j \eta(\{R_j\}) \\ &= \sum_{k=0}^{\log_{k_{accum}}(cr)} max_j \eta(\{R_j\}) \end{aligned} \tag{10}$$

Assuming that $|\{R_j\}|$ is independent of cr, we get for the average case:

$$t_a = O(\log(cr)\eta(\max_j\{R_j\})) \tag{11}$$

Summing the latencies, we get the following expression for the total latency:

$$\begin{aligned} t_{d,p,a} &= t_d + t_p + t_a \\ &= \log_{k_{distr}}(|N_{distr}|)t_{lat,distr} + \\ &\quad \max_{j \in \delta(p)} t_{proc,j} + \\ &\quad \sum_{k=0}^{\log_{k_{accum}}(cr)} max_j \eta(\{R_j\}) \end{aligned} \tag{12}$$

Now recalling that N denotes the set of all nodes, and $|N|$ is the cardinality of that set, we can state that $cr = O(|N|)$. Thus, the total time spent for solving a problem can be expressed as:

$$\begin{aligned} t_{d,p,a} &= O(\log(|N|) + c + \log(|N|)) \\ &= O(\log(|N|) \end{aligned} \tag{13}$$

Thus, the latency of problem solving is logarithmic in the number of nodes in the architecture. This bound could be found to be even tighter. However, the latency have no influence on the overall processing capacity as long as the query rate λ is lower than the system limit, $k_{proc}r$.

7 Two-Fold Linear Scalability

Given the architecture described in the previous section, it is easy to prove the claims made in Section 1 :

- The number of nodes is scalable with respect to $\sigma(D)$.
- The number of nodes is scalable with respect to $|\{P_i, \mathcal{P}_i, t\} \in P, a < t < b\}|$, that is, the number of problems arrived in any time interval.

In order to do this, an algorithm will be described that shows how to design a system that can handle a given D and P.

The algorithm consists of the following steps :

1. The actual number of processing nodes needed to handle the data D depends on the equivalence relation $\sim$ — due to the packing process described in Section 5.6, the nodes will not necessarily take full advantage of their processing capacity. However, this can be achieved by choosing a sensible $\sim$ — in the extreme case, the relation $\sim = \{(d,d), d \in D\}$ can be chosen. The number of processing nodes in each column will then be given by

$$r = \left\lceil \frac{\sigma(D)}{\sigma_{max}} \right\rceil . \tag{14}$$

2. The number of columns necessary to handle the problems in P depends on the $\mathcal{P}_i$ part of the problems $p = \{P_i, \mathcal{P}_i, t\}$. In the worst case all processing nodes need to deal with all problems, and in this case the number of columns is given by

$$c = \left\lceil \frac{\lambda}{k_{proc}} \right\rceil . \tag{15}$$

3. The total number of processing nodes is now given by

$$|N_{proc}| = cr = \left\lceil \frac{\lambda}{k_{proc}} \right\rceil \left\lceil \frac{\sigma(D)}{\sigma_{max}} \right\rceil \tag{16}$$

4. The number of nodes in the problem distribution trees will depend on the factor k_{distr}. In the worst case, if $k_{distr} = 2$, then

$$|N_{distr}| = N_{proc} - k_{distr_0} \tag{17}$$

The constant k_{distr_0} should be chosen according to fault tolerance requirements.

5. Similarly, in the worst case,

$$|N_{acc}| = N_{proc} - k_{distr_0} \tag{18}$$

The maximum number of nodes necessary to handle the specified requirements is therefore given by

$$3 \left\lceil \frac{\lambda}{k_{proc}} \right\rceil \left\lceil \frac{\sigma(D)}{\sigma_{max}} \right\rceil - 2k_{distr_0} \tag{19}$$

The scalability claims are thus proven by showing a linear relationship between both λ characterizing the incoming request rate and $\sigma(D)$ characterizing the data volume.

8 Using DPA for a Search Engine

The vast information volumes a search engine is determined to deal with certainly requires the ability to scale a solution. We shall now employ the FAST DPA framework for search engine applications.

8.1 Search engine characteristics

From [13] we learn the linear characteristics of the FAST Search engine. We also see the constraints from hardware characteristics determines sweet-spots for search engine price/performance. So, thus it is reasonable to assume the following scaling and performance attributes of a search engine node:

$$k_{proc} = O(|Q|D) \tag{20}$$
$$t_{proc} = O(|Q|D^{\alpha}) \tag{21}$$

where $|Q|$ denotes the length of the query. The processing time is linear in query length and sub-linear ($\alpha \leq 1$). However, when in the role of maintaining an SLA, one will assume that processing time is invariant by letting $|Q| = \overline{|Q|}$.

In a practical search engine setting, we will use the same physical nodes to distribute queries (problems) and accumulate results. The collapsed distribution and accumulation nodes are called *dispatchers*. A possible system is illustrated in Figure 5

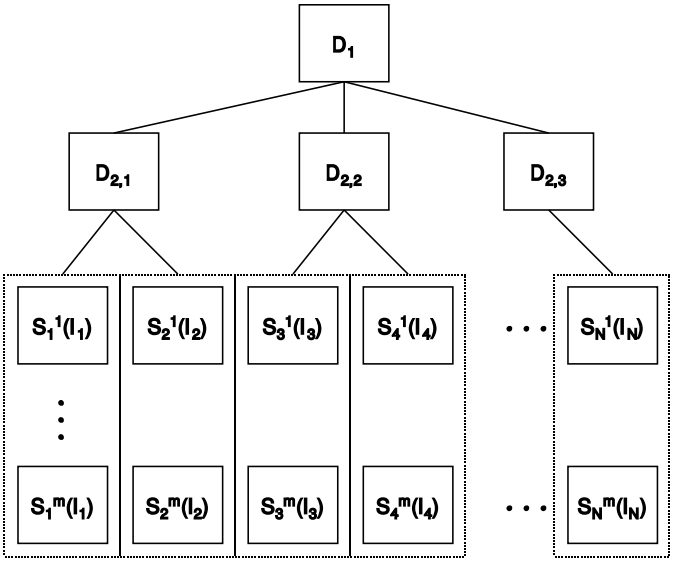


Figure 5. Cluster overview

8.2 Scaling a Search Engine

Given the definitions above, we will now describe a possible search engine configuration using DPA. Within the DPA framework, we will then need to define the following:

- The function $\phi : N_{proc} \rightarrow \mathcal{P}(\mathcal{P})$ mapping the dataset D onto processing nodes.
- The function $\delta : P \rightarrow \mathcal{P}(N_{proc})$ mapping a query into a set of nodes to answer the query.
- k_{distr}, the fan-out grade of the dispatching system.

This paper will not focus around ϕ, we are assuming that the number of documents being searched is high, and that a uniform distribution of documents is done in $\phi = \phi_{\mathbb{N},uniform}$.

Assuming that N_{proc} is organized in a set of processing arrays, each array holding a complete portion of the data, and the δ function is defined as such:

- $\delta_1 = min_{i \in [1,r]} |\gamma(N_i)|$ where $\gamma(N_i)$ denotes the pending queries in column N_i, and $|\gamma(N_i)|$ is the number of pending queries in that column.

- $\delta_2 = N_i$ — thus, select all nodes in a column to perform the search, ensuring the search will cover all data D.

By following the algorithm outlined in the previous section, we construct architecture for different document corpus sizes.

The δ scheduling function could be further improved by utilizing the knowledge about the performance characteristics of the search node. Assuming that Equation 21 holds, the pending workload could be estimated as proportional to the accumulated number of terms in pending queries for a given node.

So, let $\gamma(N_i)$ denote the set of queries pending for a given column, N_i, and let $\gamma(N_i)_j$ denote a single query in the set. Then we can define

$$\delta_1^* = min_{i \in [1,r]} \sum_{j=0}^{|\gamma(N_i)|} |\gamma(N_i)_j| \tag{22}$$

And keep δ_2 as defined above. Then $\delta\,(p) = \delta_2\,(p, \delta_1^*\,(p))$ This is now approaching a *weighted round-robin* scheduling algorithm [2].

8.3 Evaluation

To evaluate the performance and scaling characteristics of the DPA based web search system, we construct multiple clusters from the algorithm outlined in Section 7.

We use the same nodes as in [13], namely:

CPU	Dual 2.8Ghz Pentium Xeon
Memory	1GB Main memory
Disk	12 x 18Gb (10K rpm)
Disk bus	2xUW-SCSI channels

The number of documents per node for these experiments was fixed to be 5,2M randomly selected HTML pages. We wish to study two things, namely the query capacity of the system as we increase the number of nodes, and we wish to study the average responsetime as a function of the dispatching and result accumulation structures. Furthermore, we choose a fanout of 8, $k_{distr} = 8$.

So, we conduct two experiments on the different configurations:

1. Apply a sample query log, measure the maximum query capacity given that the average response time must be lower than 300ms.

Search nodes	max qps	ave searchtime, $\lambda = 400$
1	419	0.073
2	420	0.081
3	420	0.086
4	420	0.096
5	419	0.101
6	420	0.115
8	420	0.129
12 (2x6)	419	0.125
16 (2x8)	420	0.138
20 (7+7+6)	420	0.135
24 (3x8)	419	0.144
28 (4x7)	420	0.140
32 (4x8)	420	0.148
36 (6x6)	420	0.151

2. Apply a fixed rate of incoming queries, and measure the average latency for different configurations.

The query logs are used in a simulation tool that issues queries characterized by a Poisson distribution as defined in Equation 1 with a given λ (queries per second).

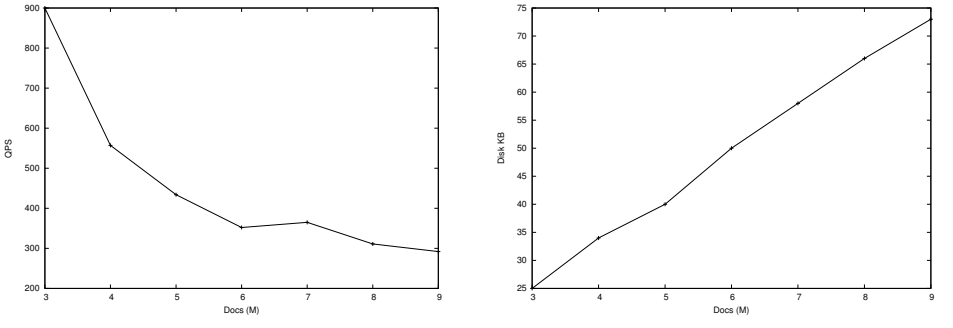
In Table 8.3 and Figure 6 the different configurations and the results of the benchmarking is shown.

From Equation 13 we expect that capacity should remain unchanged as we add nodes to the system, and that the latency should be a logarithmic relation to the number of nodes. The graph in Figure 6 shows some steps which are caused when adding levels of dispatching nodes, and having less than 8 leaf nodes for each dispatcher.

9 Conclusions and Further Work

In this paper we have outlined a general architecture for shared-nothing parallel systems that will scale linearly in processing capacity and data volume. We furthermore instantiate this architecture for the FAST Web Search system, and the experiments show that the scaling characteristics are as expected.

There are a lot of points where this architecture can be improved. The fault-tolerance discussed in this paper is very binary, and the need for proper



(a) QPS (b) ST

Figure 6. Performance metrics for different number of searchnodes

service degradation algorithms as well as better scheduling algorithms and experiments over there are apparent.

References

[1] Anderson, T. E., Culler, D. E., and Patterson, D. A. A case for NOW (Networks of Workstations). *IEEE Micro 15*, 1 (Feb. 1995), 54–64.

[2] Aron, M., Sanders, D., Druschel, P., and Zwaenepoel, W. Scalable content-aware request distribution in cluster-based network servers. In *Proceedings of the USENIX 20 Annual Technical Conference* (San Diego, CA, June 2000), pp. 323–33.

[3] Bowman, C. M., Danzig, P. B., Hardy, D. R., Manber, U., and Schwartz, M. F. The Harvest information discovery and access system. *Computer Networks and ISDN Systems 28*, 1–2 (1995), 119–125.

[4] Bowman, C. M., Danzig, P. B., Manber, U., and Schwartz, M. F. Scalable Internet resource discovery: research problems and approaches. *Communications of the ACM 37*, 8 (1994), 98–107.

[5] Brewer, E. A. Delivering high availability for inktomi search engines. In *SIGMOD 1998, Proceedings ACM SIGMOD International Confer-*

ence on Management of Data, June 2-4, 1998, Seattle, Washington, USA (1998), L. M. Haas and A. Tiwary, Eds., ACM Press, p. 538.

[6] DOUGHERTY, E. R. *Probability and statistics for the engineering, computing, and physical sciences.* Prentice Hall, 1990.

[7] FLYNN, M. J. Very high-speed computing systems. *Proceedings of the IEEE 54* (1966), 1901–1909.

[8] FOX, A., GRIBBLE, S. D., CHAWATHE, Y., BREWER, E. A., AND GAUTHIER, P. Cluster-based scalable network services. In *Symposium on Operating Systems Principles* (1997), pp. 78–91.

[9] JUDSON, T. W. *Abstract Algebra.* PWS Publishing Company, 1994.

[10] MACLEOD, I. A., MARTIN, P., AND NORDIN, B. A design of a distributed full text retrieval system. In *Proceedings of the 9th annual international ACM SIGIR conference on Research and development in information retrieval* (1986), ACM Press, pp. 131–137.

[11] MELNIK, S., RAGHAVAN, S., YANG, B., AND GARCIA-MOLINA, H. Building a distributed full-text index for the web. In *World Wide Web* (2001), pp. 396–406.

[12] RIBEIRO-NETO, B. A., AND BARBOSA, R. A. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the third ACM conference on Digital libraries* (1998), ACM Press, pp. 182–190.

[13] RISVIK, K. M., AND EGGE, T. The **fast** search engine kernel and its performance characteristics. Tech. rep., Fast Search & Transfer ASA, 2002.

[14] SUEL, T., MATHUR, C., WU, J., ZHANG, J., DELIS, A., KHARRAZI, M., LONG, X., AND SHANMUGASUNDERAM, K. Odissea: A peer-to-peer architecture for scalable web search and information retrieval, 2003.

[15] TOMASIC, A., AND GARCIA-MOLINA, H. Query processing and inverted indices in shared: nothing text document information retrieval systems. *The VLDB Journal 2*, 3 (1993), 243–276.

Multi-tier Architecture for Web Search Engines

Knut Magne Risvik
Overture Services AS
P.O.Box 4452 Hospitalsløkkan
NO-7418 Trondheim, Norway
knut.risvik@overture.com

Yngve Aasheim
Overture Services AS
P.O.Box 4452 Hospitalsløkkan
NO-7418 Trondheim, Norway
yngve.aasheim@overture.com

Mathias Lidal
Overture Services AS
P.O.Box 4452 Hospitalsløkkan
NO-7418 Trondheim, Norway
mathias.lidal@overture.com

Abstract

This paper describes a novel multi-tier architecture for a search engine. Based on observations from query log analysis as well as properties of a ranking formula, we derive a method to tier documents in a search engine. This allows for increased performance while keeping the order of the results returned, and hence relevance, almost "untouched". The architecture and method have been tested large scale on a carrier-class search engine with 1 billion documents. The architecture gives a huge increase in capacity, and is today in use for a major search engine.

1 Introduction

Search engines are becoming a very important application for navigating the web. Search engines face huge challenges trying to keep up with the web dynamics. The web is growing exponentially, and it is becoming more and more dynamic. Risvik and Michelsen[11] discusses the dynamics of the web, and the challenges it imposes on the large search engines.

The ability to consistently deliver excellent result relevance is probably the single most important factor for user satisfaction in a web size search system.

When we disregard the fixed costs of maintaining a huge index, the cost of search execution is close to linear to the size of the index searched;

doubling the amount of information in the index will generally double the cost of query execution.

Search engines of this size are usually distributed applications, using replication and partitioning to scale to the desired number of documents. We intend to utilize properties of query distributions and ranking formula contributions to optimize which nodes in the mesh to use for each query.

2 Preliminaries

Most practical and commercially operated Internet search engines are based on a centralized architecture that relies on a set of key components, namely Crawler, Indexer, Searcher and Dispatcher. This architecture can be seen in systems including AltaVista [2], Google[3], and the FAST Search Engine [1], and is illustrated in Figure 1.

Definition 10 Crawler. *A crawler is a module aggregating documents from the World Wide Web in order to make them searchable. Several heuristics and algorithms exist for crawling, most of them based upon following links in hypertext documents.*

Definition 11 Indexer. *A module that takes a collection of documents or data and builds a searchable index from them. Common methods are inverted files, vector spaces, suffix structures and hybrids of these.*

Definition 12 Searcher. *The searcher is working on the output files from the indexer. The searcher accepts user queries from the dispatcher (defined below), executes the query over its part of the index, and returns sorted search results back to the dispatcher with document ID and the relevance score (defined below).*

Definition 13 Dispatcher. *The dispatcher receives the query from the user, compiles a list of searchers to execute the query, sends the query to the searchers and receives a sorted list of results back from each searcher. For each result it receives a unique document ID, and the relevance score. The hits from the searchers are then merged to produce the list of results with the highest relevance scores for presentation to the user.*

Search engines generally operate by producing a result set for each query (a set of documents that matches the query), and then ranks the documents by using a formula to compute a relevance score for each entry in the result set with respect to the query being executed and sorting the documents by their relevance score.

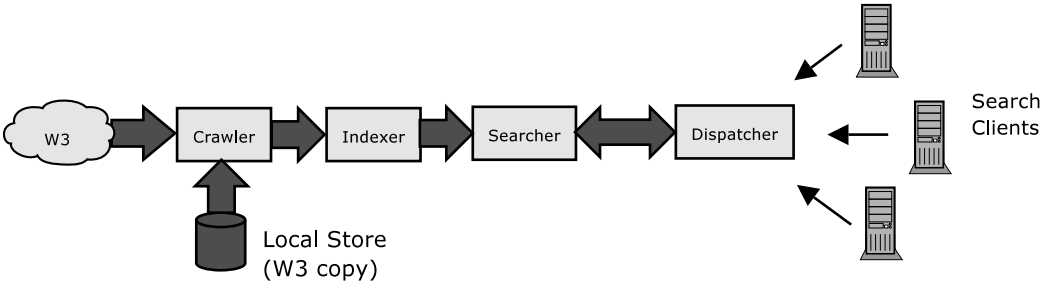


Figure 1. Search Engine reference model

Definition 14 Relevance score *Upon a given query, q, the search engines give back a sorted list of results. Each document in the result has a relevance score $R_{d,q}$ (Relevance of document d with respect to query q). The relevance score can be further broken down into a static component Rs_d that is independent of the query, and a dynamic component $Rd_{d,q}$ dependent on the query. The total relevance score formula is then*

$$R_{d,q} = Rs_d + \alpha Rd_{d,q} \tag{1}$$

The constant α is used to balance the weight of the static and the dynamic relevance scores.

For a web search engine, utilizing link structure and HTML semantics, the relevance score is usually derived from several different contributing attributes:

- Static relevance score for the document (link cardinality, page quality).
- Superior parts of the document. (Titles, metadata, document headers)
- Authority (external references, and the "level" of these).
- Document statistics (by e.g. term frequency in the document, global term frequency and term distances within the document).

3 Related Work

From several studies of query logs ([12], [5]) and also supported by our own analysis presented in this paper, there is a highly skewed distribution of

unique queries on a typical web search engine. Both [12] and [5] reports that the top 25 queries count for more than 1% of the total query volume.

A vast amount of work has been conducted on cache algorithms for web search engines, and for information retrieval systems in general. In [6] we see a study of different cache-replacement schemes for web search engines, testing both standard LRU, segmented LRU and a predictive system based on session analysis named PDC (Probability Driven Cache).

The results are promising, one sees more than 50% hit ratio of the caches, clearly being a very important optimization tool for any serious web search engine.

On a different perspective, there has been several studies on how to partition a large dataset in an information retrieval system and to provide some sort of a broker algorithm to select a subset of these partitions for searching with the intent of providing an approximation of the search results from searching the entire collection.

In [7] and in particular [9] we see studies using locality analysis of the query log to compute a partial replica of the entire collection and to use a broker (to multiple InQuery servers) server to select the replica. In [8] we see the same technique applied on a large dataset.

4 This Work

In this work we propose an architecture for a scalable web search engine that uses multiple log and relevance analysis to build tiers of documents. The architecture is novel to prior work in the sense that each tier is disjoint from the others (as opposed to partial replication where the smaller sets are subsets of the main index). Furthermore we deploy a fallthrough algorithm to allow for queries to "fall" from one tier to another based on analysis of the query and the results from the given tier.

We run experiments on 3 different architectures inspired by results from partial replication and query cache analysis as well as the ranking function itself. The experiments provide evidence that these architectures offer a significant increase in query capacity over regular caching.

5 Target Architecture

The analysis described in this paper are all based on the same conceptual architecture. We will describe the architecture in two levels, first the cluster concept, then the tiering concept.

Documents	QPS	disk KB	disk TPS	CPU idle
3M	900	25	2000	45%
4M	557	34	1720	41%
5M	434	40	1560	42%
6M	352	50	1350	43%
7M	365	58	1450	35%
8M	311	66	1315	37%
9M	292	73	1310	35%

Table 1. Data volume performance

5.1 Basic Elements

The basic element in the target architecture is a search node. A search node holds a partition, an index for a fraction of the entire database, and allows for searches in that partition. Letting I denote the entire database, a search node S_j typically holds I_j.

5.2 Performance for different data volumes

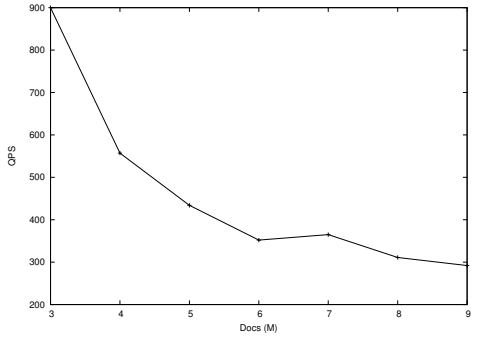
In this paper we will use the FAST Search engine kernel. This engine has performance characteristics that, within certain boundaries, can be considered linear with the number of documents indexed per node. Some performance numbers are shown in Table 1, and illustrated in Figure 2, where QPS denotes the number of queries per second, disk KB denotes the number of kilobytes read per query, disk TPS indicates the number of disk transactions per second, and CPU idle is the percentage of CPU slices idle during testing.

All these tests were conducted on a dual CPU (2x2.4Ghz) Pentium 4 with 1GB of main memory and 12 x 36 GB disk drives. The benchmarking was done with 50 parallel clients, and maintaining average response time of a query below 0.5 seconds, the standard customer SLA (service level agreement).

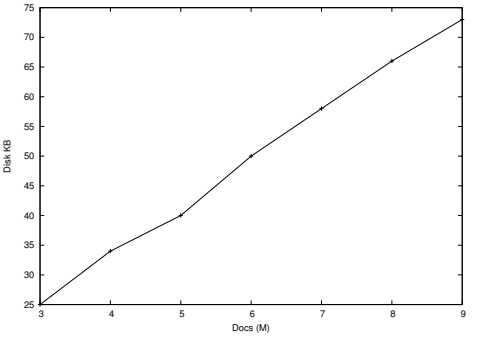
Now, letting $QPS(S_j)$ denote the query capacity of search node S_j in queries per second, and furthermore assuming that the average length of queries are constant, the following relation is given:

$$QPS(S_n) = O(|I_n|) \tag{2}$$

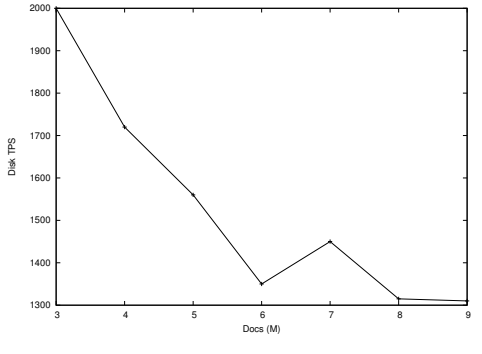
The FAST Search kernel employs several types of performance optimizations that is required for this approximation to be applicable. See Figure 2



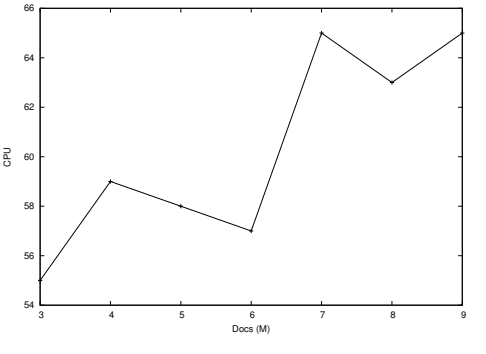
(a) QPS



(b) KB



(c) TPS



(d) % Free CPU-time

Figure 2. Performance metrics for different document sizes

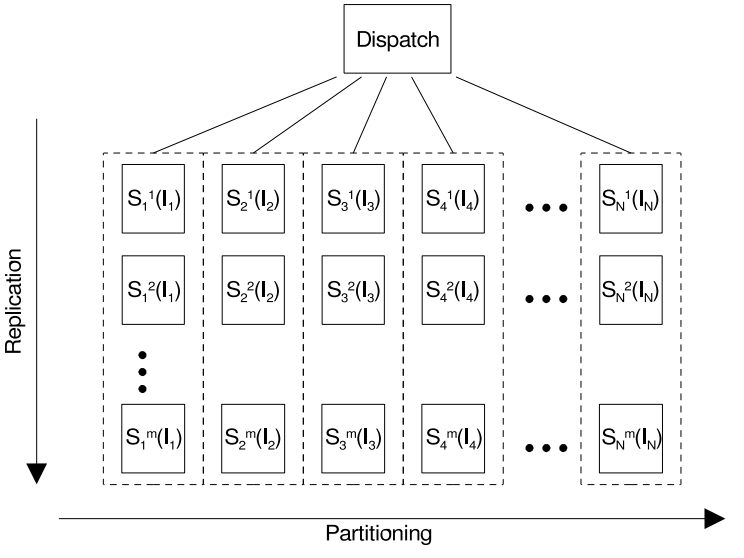


Figure 3. Cluster overview

(a) for an illustration of this correlation.

5.3 Basic Scalability

Now, let a cluster be a collection of search nodes that are grouped in rows and columns. Partitioning and replication are then used to create linear scalability in size and query rate. The basic principles of this architecture is also covered by [10].

5.3.1 Data volume scaling by partitioning

Let each search node hold a partition of the index, I_j of the entire index I. Each search node queries its partition of the index upon a request from a dispatcher. The dispatcher sends an incoming query to a set of search nodes such that all partitions $I_1 \ldots I_n$ are asked. The results are merged and a complete result from the cluster is generated. We use the notion of a *row* to name a set of search nodes making up for all partitions of the entire index. Thus, by partitioning the data, we create scalability in data volume, as illustrated in Figure 3. Given Equation 2, and assuming that we split the data into n partitions, the search time on each node would be $O(|I|/n)$. n is selected based upon hardware and software criteria, but varies between 1M and 20M documents.

A limiting factor here is the merging of results that needs to take place in the dispatcher. A heap-based algorithm typically gives $O(mlogn)$ to merge m results from n sources. m is typically low due to the interactive nature of a search system.

5.3.2 Performance scaling by replication

By replicating each of the search nodes, we are able to increase the query processing rate for a given partition of the index. Letting S_i^j denote a search node in an nxm cluster, all search nodes $S_i^1 \ldots S_i^m$ holds index partition I_i. Thus, the dispatcher can rotate between m nodes for each index partition when selecting a set of search nodes to handle an incoming query.

The dispatcher handles this by a round-robin fashion algorithm. By adding rows, performance will increase proportionally to the the capacity of a single row.

The entire architecture in Figure 3 illustrates the organization into rows and columns, and the role of the dispatcher.

5.4 Tiered Architecture

Based on the search node clusters described above, we now intend to build a tiered architecture of search nodes.

Conceptually, the tiered architecture groups documents into tiers $T_1 \ldots T_m$. Each query starts off in tier 1, and a fallthrough algorithm FTA selects whether the query shall continue execution into consecutive tiers and how results are to be merged.

Thus, there are three elements to the tiered architecture:

Definition 15 Tier mapping. *Given a set of documents D, the function $\mathcal{F}$ takes a document D_i with a vector of properties, $\vec{P_{D_i}}$ and maps it into a given tier T_j.*

Furthermore, the algorithm to define the fallthrough needs to be defined:

Definition 16 Fallthrough - *FTA. Each query starts off in the lowest tier. The fallthrough algorithm, FTA, determines the path of the query in the set of tiers based on criteria such as relevance scores and number of hits in the result set. The FTA is responsible for determining how many results from each tier can be used before the next tier must be consulted.*

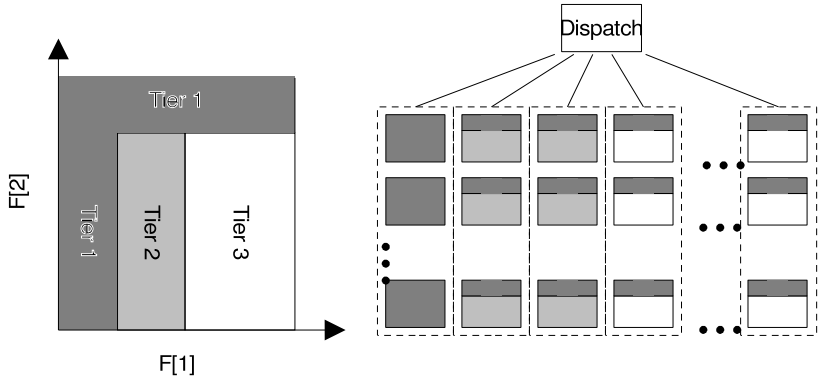


Figure 4. Mapping from a document space into tiers of search nodes

When the FTA decides that a new tier must be consulted, the relevance score for the new results will be considered when merging results to produce the final list of results to be presented to the user.

In Figure 4 an example is illustrated. Given that the documents can be viewed in a two dimensional feature space, we can illustrate the document space as in the first figure. Then tiering takes place by applying $\mathcal{F}$ to map each document to the desired tier. Then mapping can be done into search nodes as shown.

Figure 4 also illustrates that documents can be divided into sections for tiers. For instance, one could define tier 1 to be the superior context (titles and anchors) of all documents, and tier 2 to be the body context of a small selection of the documents. Thus, a search node will hold data for multiple tiers, but without duplicating the data stored.

The tiered architecture has an impact on performance when the FTA is able to reduce the number of search nodes involved in a query, and the appearing results have little or no deviation from what you would get by searching all nodes. Of course, this implies an interactive system, where the top 10–100 results are the interesting ones.

6 Query locality

We aim to understand the locality of queries and to estimate the confidence of these locality figures. By locality we mean to what degree the same queries are received multiple times, both over short and long time-periods.

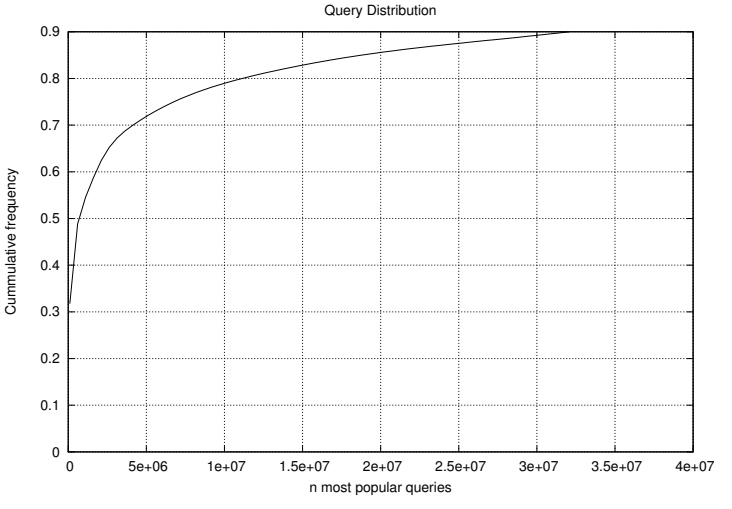


Figure 5. Query frequency distribution

Provided a significant number of queries are duplicates, we can build a tiering mechanism that would have a significant positive impact on performance of the entire search system. The queries we examine are gathered from the query log of AllTheWeb.com. The query log is a log showing the query-string, along with the date and time of the query, and some more information which we do not use at this time.

6.1 Query Distribution

First we study the query distribution. By folding the queries into unique queries, and sorting the bins based on query frequency, we get a sorted distribution of queries.

The queries we examine are collected from the queries executed by users of AllTheWeb.com. We analyze five different query-sets, collected from the following time-periods:

- 3. of January 2002
- 15. - 21. (3. week) of January 2002
- 1-31. of January 2002
- 11. of February 2002
- 11. of March 2002

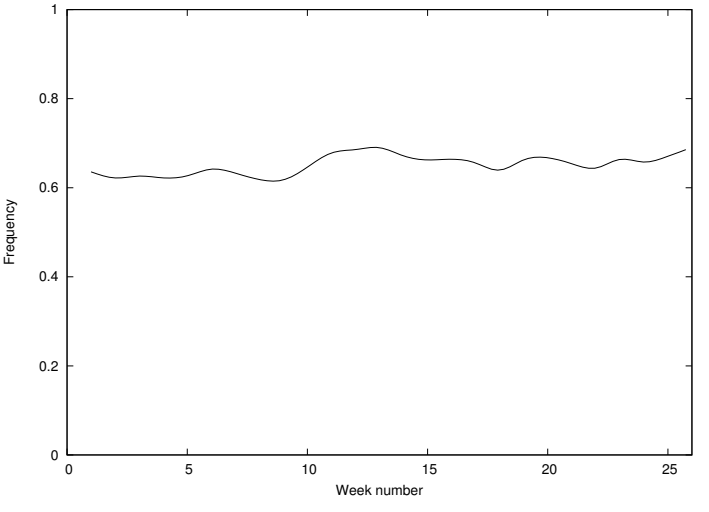


Figure 6. Frequency of top 1 mill. queries first half of 2002

Time period	Num. queries	unique queries	num. queries within top 1 million
3. Jan 2002	19 846 936	4 560 172	15 239 298 (76.8%)
15.21. Jan 2002	132 741 183	22 590 276	83 108 836 (62.6%)
Jan. 2002	573 394 981	81 721 575	307 434 094 (53.6%)
11. Feb 2002	20 062 879	4 300 090	15 678 972 (78.14%)
11. Mar 2002	20 714 809	4 311 001	16 518 754 (79.74%)

Table 2. Summary of querysets

These query-sets enable us to analyze the queries from both overlapping and disjoint time-periods, which should give a good indication of the level of locality of the queries. These querysets are summarized in Table 2, showing the total number of queries, the number of unique queries and the number of queries made up of the 1 million most popular unique queries.

We now want to analyze how the most popular queries do in the overall frequency picture. Here we used the largest query-set, from January 2002. We collected the cumulative frequency of the n most popular queries. The distribution of this frequency is shown in Figure 5.

We observe that the query distribution is strongly logarithmic, thus we have an exponential growth in the number of queries to hold a given percentage of the total queries.

This distribution is fairly constant over time, as shown in Figure 6 which shows the frequency of the 1 million most popular queries per week for the first half of 2002.

Furthermore, we count the number of unique queries (i.e. queries that are only asked once), and we find that 41,748,880 queries out of the 573,394,981 queries recorded in January only occur once. This is 7.28% of the queries.

6.2 Temporal overlap

Here we examine the overlap between queries from different time-periods, focusing on the 1 million most popular queries.

Letting Q_1 and Q_2 denote the set of queries for each consecutive period, the overlap is computed as:

$$O(Q_1, Q_2) = \frac{|(Q_1 \bigcap Q_2)|}{|(Q_1 \bigcup Q_2)|)} \tag{3}$$

Now, letting O_n denote the overlap of the n most frequent queries from Q_1 and Q_2, we compute the overlap for different number of unique queries.

First we examine the overlap between the query-sets from January. This will give an indication of how representative the results of a single day or week is. Next we examine the overlap between the three different days, as well as the overlap between the queries from January and those from one day in February and March. This will give an indication of how much the queries differ over time. Table 3 shows the overlap between the different query-sets.

	1 day (Jan)	1 day (Feb)	1 day (Mar)	1 week	1 month
1 day (Jan)		82.04%	80.70%	85.06%	84.53%
1 month	86.65%	84.72%	84.53	92.54%	

Table 3. Temporal overlap

6.3 Locality

These results show that there is a high degree of repetition in the queries, with only 7.28% of the queries being asked just once. There is also a very high temporal overlap, with 80 - 85% overlap on single days in different months, when looking at the top 1 million queries. These queries account for 75-79% of all queries these days.

7 Applications of the tiered architecture

We have evaluated three different architectures and document distribution functions, $\mathcal{F}$, and different fallthrough algorithms. For the different architectures, we compute the quality of the architecture as two numbers based upon the difference between its search results compared with the search results from a predefined reference system, typically a single-tier system.

The Fallthrough algorithm is basically equal for all three architectures, and consist of 5 parameters:

- *Hitlimit* - Specifies the maximum number of hits to be used from a tier before fallthrough to the next tier is forced.
- *Percentlimit* - The maximum percentage of the hits from this tier that may be used before fallthrough to the next tier is forced.
- *Ranklimit* - Used together with the *Termranklimit*. See *Termranklimit*.
- *Termranklimit* - If the relevance score of the hit being considered is less than the *Ranklimit* plus this value times the number of terms in the query, then fallthrough to the next tier is forced.
- *MinUsableHits* - The number of hits that must pass the above criteria for a given tier in order not to do an immediate fallthrough to the next tier. This number is typically the number of results that is presented to the user on a result page. The rationale for using this rule is that if it is known that we will have to fall through in order to produce the

Tier jump	Hitlimit	Percentlimit	Ranklimit	Termranklimit	Minhits
1–2	1000	10	200	0	0
2–3	8100	30	0	0	100

Table 4. Basic FTA

number of hits most often requested, we should perform the fallthrough immediately. We do not want to apply this rule on a non-constant value (such as the number of hits really requested by the user), in order to ensure that we produce consistent results.

```
boolean FallThrough(Results r,
                    FallthroughConfig c,
                    int hitNo,
    int queryTerms) {

    // Verify HitLimit rule
    if (c.hitLimit < hitNo)
        return true;

    // Verify PercentLimit wrt. requested hit
    // and MinUsableHits parameter
    if (max(hitNo, c.minUsableHits) >
       ((r.numResults * c.PercentLimit) / 100))
        return true;

    // Verify RankLimit and TermRankLimit wrt.
    // requested hit and MinUsableHits parameter
    if (r.Result[max(hitNo, c.minUsableHits)].
       relevanceScore <
       (c.rankLimit + c.termRankLimit *
        queryTerms))
        return true;

    return false;
}
```

The parameter values for the basic FTA is defined in Table 4.

For the multi-tier concept to work, we need to have higher performance on tier 1, which is going to execute all queries, than on tier 2 which will

only execute those queries that fall over from tier 1. Tier 3 will, in turn, only execute queries that fall over from tier 2. The extra capacity on the lower tiers may be achieved either by replicating the columns on this tier, or by reducing the number of documents on each node, in order to achieve higher throughput on each node. From an operations point of view, the latest option is more flexible, hence this is the preferred approach.

As already stated in this paper, the performance of a node is considered to be linearly dependent on the number of documents on the node. For the tests runs being described in this paper, we had 1.5M documents indexed on each tier 1 node, 6M documents on each tier 2 node and 10M documents index on each tier 3 node.

7.1 Evaluation

To evaluate the different multi-tier configurations, we randomly select 100000 unique queries from the list of queries entered by users of FAST Web Search. The selected queries are then executed twice, once towards the multi-tier configuration we want to evaluate and once towards the reference system. The results from the two runs are compared to extract information about the quality of the multi-tier configuration.

We use two metrics to indicate the quality of search results in comparison with the reference system:

- **Different First hit**. The percentage of queries that return a different hit in the first position.

- **Different Top 10**. The percentage of queries for which there are one or more differences within the first ten results.

With an ideal solution, those percentages are zero or close to zero.

Combined, those two numbers are believed to give good knowledge about key attributes of the systems relevance penalty with respect to the reference system.

Furthermore, we measure the total system query capacity to understand the performance impact of the multi-tier architecture. This is done by executing a standard query log (typically 100K to 1M queries) onto the search cluster and measure the query production rate at the standard SLA. Then the speedup ratio between the reference system and the candidate system is computed.

Tier 1	30M documents
Tier 2	360M documents
Tier 3	610M documents

Table 5. 1D Multi-tier configuration

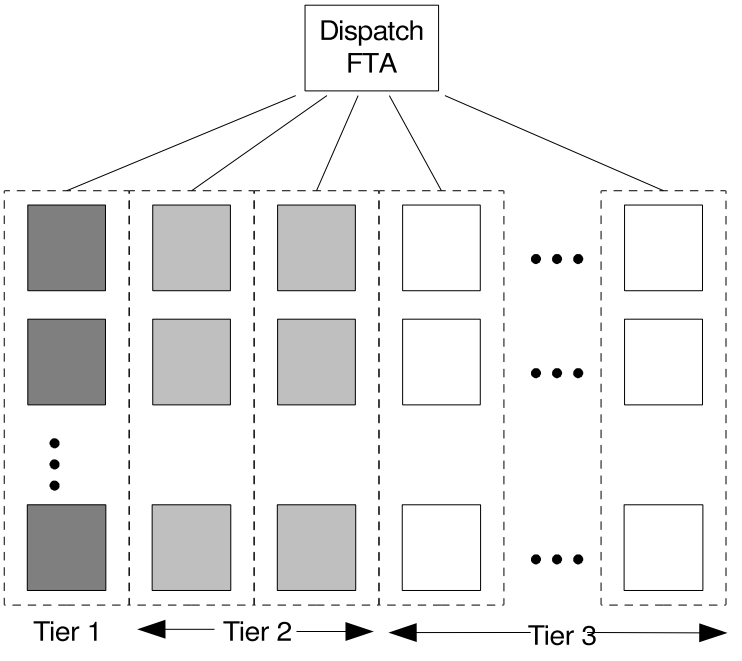


Figure 7. 1-dimensional MT system

The experiments are conducted on a sample collection containing 1 billion documents. The number of documents to be index on each tier have been predefined, and are the same for all three configurations to be tested.

7.2 1-dimensional multi-tier configuration

This is a plain three-level multi-tier configuration, with all documents being distributed by the static relevance score. The 30M top documents are then mapped to tier 1, the 360M next documents to tier 2, etc. The breakdown of documents into tiers is shown in Table 5.

The system is illustrated in Figure 7, using the standard *FTA* as defined in Table 4.

Tier 1	30M documents (5M tierlocked)
Tier 2	360M documents
Tier 3	610M documents

Table 6. 1.5D Multi-tier configuration

The reference system chosen is a plain single-tier configuration, thus searching all nodes simultaneously.

One significant problem with this configuration is that static relevance is only a part of the formula for determining the relevance score of a hit. This means that a document with low static relevance score can still be a good hit for certain queries. The next configuration is an attempt to reduce this problem.

7.3 1.5-dimensional multi-tier configuration

Again, this is a plain three-level multi-tier configuration. Before doing tier distribution, we run a query log with the 1M most common queries for a period in time. To avoid delaying the indexing process in the production system, this query log was executed towards the previous generation index is executed on the previous index. The first 20 documents returned for those queries will be indexed on the first tier.

The remaining documents are distributed according to the static relevance score, so the tier breakdown is as shown in Table 6. The system is illustrated in Figure 8, and was tested using the FTA as defined in Table 4.

Again, the reference system is a plain single-tier configuration.

7.4 2-dimensional multi-tier configuration

The tier distribution for this configuration is identical to the tier distribution for the 1.5-dimensional multi-tier configuration. The difference is the way the information is searched. With this configuration, we search the information in the high-value contexts for all documents first. If we need more hits, we continue to search the full index, using a multi-tier configuration, removing duplicates from the returned results.

Obviously, searching the high-value contexts for all documents will consume its fair share of resources available on the second and third tiers. This mean that we have capacity for processing fewer queries on those tiers when searching the full index on those nodes.

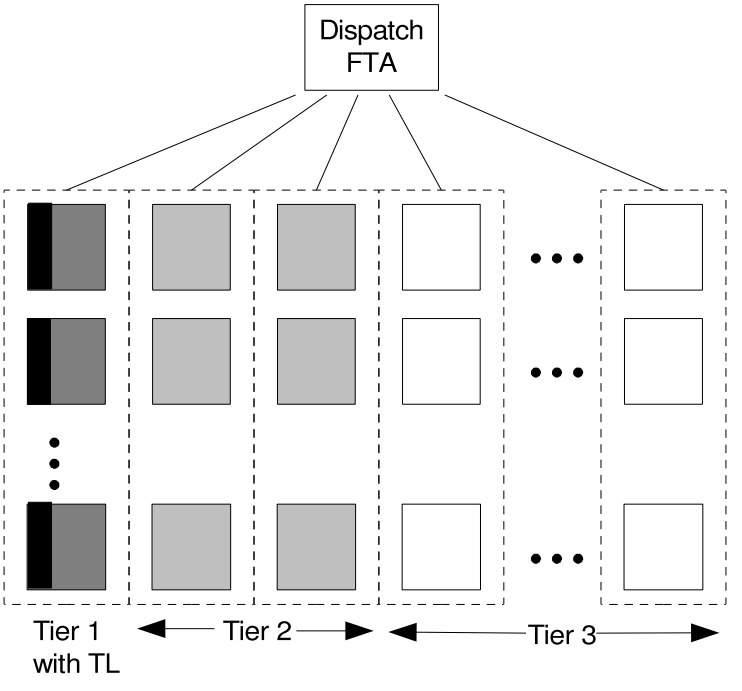


Figure 8. 1.5-dimensional MT system with tierlocking

Tier 0	1B documents - superior context
Tier 1	30M documents - body context
Tier 2	360M documents - body context
Tier 3	610M documents - body context

Table 7. 2D Multi-tier configuration

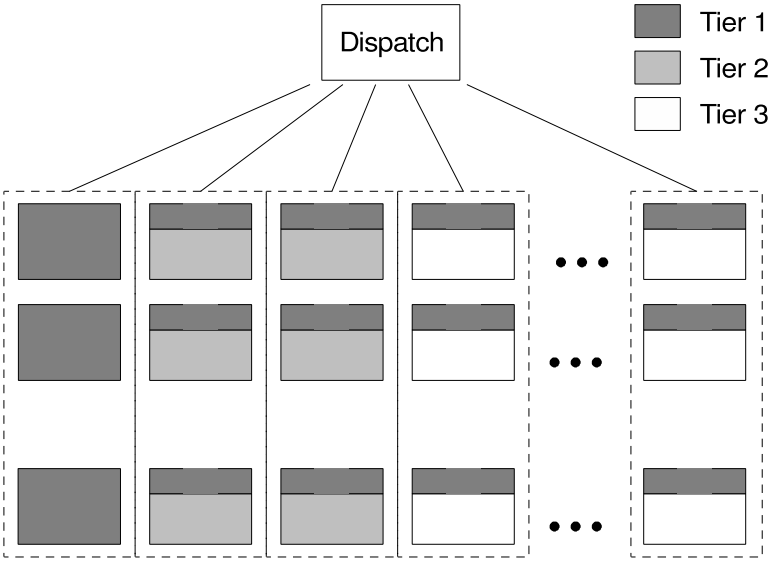


Figure 9. 2-dimensional MT system

Tier jump	Hitlimit	Percentlimit	Ranklimit	Termranklimit	Minhits
(0+1)–2	8100	50	200	0	100
2–3	8100	70	200	0	100
3–4	8100	50	0	0	100

Table 8. 2D FTA (Note that 0 and 1 are searched in parallel.)

The system has tier distribution as shown in Table 7, and it is illustrated in Figure 9. As shown, Tier 0 is introduced as the layer of superior contexts.

The FTA for this system is slightly modified from the original one (Table 4). The modified FTA is shown in Table 8.

The reference system defined for the other multi-tier configurations can not be used directly for this configuration, since this multi-tier configuration not directly emulates a plain single-tier solution. We still want to use a single-tier system, but we now need to search high-value contexts for all nodes first, then merge in results from a search in the full-text index.

This change in the reference system naturally changes the order of the results produced by the search engine. However, manual verification by editors as well as theoretical studies indicate that, given the way we currently

System	Different #1	Different #10	QPS ratio
1D Multi-tier	26.8%	62.1%	2.78
1.5D Multi-tier	15.4%	45.8%	2.71
2D Multi-tier	4.36%	17.7%	2.02

Table 9. Summary of results

compute the relevance score, this change more often improve the relevance than reduce it.

7.5 Results

A summary of the experiments is shown in Table 9. The 1D Multi-tier solution heavily relies upon high covariance between the rank of a search result and the static relevance score of the documents in the search result. This assumption is often wrong for web search queries are for most cases wrong, thus we will see big deviations from the reference system. This is mainly due to the nature of static relevance score, which is mostly based upon link cardinality. In [4] the link structure of the web is discussed.

Moving on to the 1.5D architecture, we utilize the locality within query logs to bias the creation of the first tier. This model will guarantee that the most popular queries (which accounts for a very high total number of queries) always gets the "correct" results compared with the reference system. This is clearly indicated on the results. However, the queries falling out of the "popular" group are still subjects for the same treatment" as with the 1D system.

The key observation from the 2D system is that the usage of more document properties, and another dimension in the tier space has a much higher covariance with the actual ranking function. So by letting the relevance score space be divided into static relevance score and the binary selection of superior and non-superior context, the approach has a relevance that is very appealing given the performance capabilities. Also, the use of a reference system that has proved superior to the other reference systems by editorial inspection is promoting this as a very promising architecture. The end result is more than a doubling of the performance without much sacrifice in relevance

The performance of the different configurations, as stated in the QPS ratio column of Table 9 have been measured by benchmarking the different systems to extract information about the highest possible throughput that can be achieved while complying with the standard SLA.

8 Capsule summary of FAST

Fast Search & Transfer (FAST) (http://www.fast.no/) is a Norwegian company that has been operating in the web search and corporate search segments for years. The web search division is probably most known for its search engine AllTheWeb (http://www.alltheweb.com/), and for supplying search results to portals in most parts of the world. After this paper was drafted, the web search product was sold to the American company Overture. Overture is currently in the process of being acquired by Yahoo.

9 Conclusive Remarks and Future Work

We have shown how multi-tier architecture can be used to achieve "superlinear" scaling ofweb size search engines. Sample systems are able to achieve more than double the capacity with a relatively small relevance penalty.

The testing has been restricted to a handful of configurations, but the results are still very promising. Moving along, it will make sense to do much more extensive analysis to determine the dimensions of the document space, and possibly one could use mining techniques to derive the document to tier mapping rules and fallthrough algorithms.

In order to efficiently design and tune a multi-tier search architecture, we aim to build a simulation tool that can simulate different fallthrough algorithms and different tier separation rules. Evaluation of possible performance benefits and relevance penalties with different tier sizes is equally interesting.

For a configuration being used for production purposes, one would want a system that continuously monitor the number of queries that get a relevancy penalty as described in this document.

References

[1] Alltheweb. http://www.alltheweb.com.

[2] Altavista. www.altavista.com.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th WWW Conference*, http://decweb.ethz.ch/WWW7/1921/com1921.htm, 1998.

[4] A. B. et. al. Graph structure in the web. In *Proceedings of the Ninth International World Wide Web Conference (WWW9)*, 2000.

[5] B. J. Jansen, A. Spink, and T. Saracevic. Real life, real users, and real needs: a study and analysis of user queries on the web. *Information Processing and Management*, 36(2):207–227, 2000.

[6] R. Lempel and S. Moran. Predictive caching and prefetching of query results in search engines. In *Proceedings of the Twelfth International World Wide Web Conference (WWW2003)*, 2003.

[7] Z. Lu. Scalable distributed architectures for information retrieval. Technical Report UM-CS-1999-049, , 1999.

[8] Z. Lu and K. S. McKinley. Searching a terabyte of text using partial replication. Technical Report UM-CS-1999-050, , 1999.

[9] Z. Lu and K. S. McKinley. Partial collection replication versus caching for information retrieval systems. In *Research and Development in Information Retrieval*, pages 248–255, 2000.

[10] K. M. Risvik, T. Egge, B. Svingen, and A. Halaas. Search engine with two-dimensional linear scalable parallel architecture. International Patent PCT/NO99/00155, 2000.

[11] K. M. Risvik and R. Michelsen. Search engines and web dynamics. *Computer Networks (Amsterdam, Netherlands: 1999)*, 39(3):289–302, 2002.

[12] C. Silverstein, M. Henzinger, J. Marais, and M. Moricz. Analysis of a very large altavista query log. Technical report, SRC Technical Note, 1998.

Linguistics in Large-Scale Web Search

Jon Atle Gulla
Fast Search & Transfer ASA
P.O. Box 4452 Hospitalsløkkan
NO-7418 Trondheim, Norway
Jon.Atle.Gulla@fast.no

Per Gunnar Auran
Fast Search & Transfer ASA
P.O. Box 4452 Hospitalsløkkan
NO-7418 Trondheim, Norway

Knut Magne Risvik
Fast Search & Transfer ASA
P.O. Box 4452 Hospitalsløkkan
NO-7418 Trondheim, Norway

Abstract

In spite of intensive research on linguistic techniques in information retrieval, there are still few large-scale search engines that have taken full advantage of these techniques. This paper presents the integration of various linguistic techniques in one of the largest search engines on the web. The techniques include language identification, text categorization, offensive content filtering, normalization, phrasing and anti-phrasing, and clustering. We go into some of the challenges of dealing with huge amounts of data and discuss the compromises that were needed to do this integration. Our results show that linguistics has a great potential in web search, though it is also clear there are many unsolved issues and challenges.

1 Introduction

Search engines are today important to any user retrieving information on the Internet. The explosion of documents and the lack of inherent structures or directories on the Internet have made in increasingly difficult for users to retrieve the desired documents. When they get confused by the multitude of information available, they turn to large-scale search engines like AllTheWeb, Google and AltaVista for help. 85% of Web users today claim to be using search engines or some kind of search tools to find specific information of interest [6]. According to Economist Magazine's survey from 2000 [11], more than 70% of American households use search engines on a

weekly basis. In fact, almost a third of all web sessions these days involve search engines [18].

The importance of search engines has also been noted by information providers in the Internet market. When Forrester in September 2001 interviewed IT managers in 49 Global 3500 companies that were planning or were implementing portals, search was considered the most important feature of their portal. Also, large companies increasingly need sophisticated search technologies to handle their exhaustive content directories [7].

However, web search is faced with a number of challenges. Large volumes of data are distributed over many computers and platforms [15]. There are dead links, dynamic pages, and relocations disturbing the search, and a large share of the documents are poorly structured or even redundant. The proliferation of languages and alphabets on Internet also makes it hard to introduce language-specific or culture-specific techniques to improve the situation. Still, there is usually enough information on Internet to answer even the weirdest question. The problem is not primarily to find documents, but to find documents that are relevant to the query and ranked in a meaningful way.

Linguistic techniques in information retrieval systems address the relevance of documents returned. They allow us to abstract away from the exact words used in the documents and queries and put more emphasis on the content of these representations. As such, they should help us close the gap between the user's expressed query and the representation of relevant documents. Although the experiences with linguistic techniques in information retrieval are somewhat divided, few doubt their potential for improving the relevance of the documents retrieved ([1][10][2]).

Fast Search & Transfer (FAST) has one of the biggest search engines on the market, and its engine is integrated in portals like Lycos and T-Online as well as in enterprise solutions for IBM, eBay, and other large international companies. FAST is offering their web search engine as an OEM service to portals, as well as the showcase portal AllTheWeb (www.alltheweb.com). The Search engine holds a number of innovative linguistic techniques in their search engine on AllTheWeb (www.alltheweb.com). Offering search facilities on a global basis, the company has opted for approaches that are cross-lingual and allow continuous updates with new languages and improved dictionaries.

This paper discusses the introduction of language identification, offensive content filtering, normalization, and query transformation in FAST. Rather than evaluating the effect of each technique, we look into the particular problems that show up in large-scale search engines for the web. Whereas

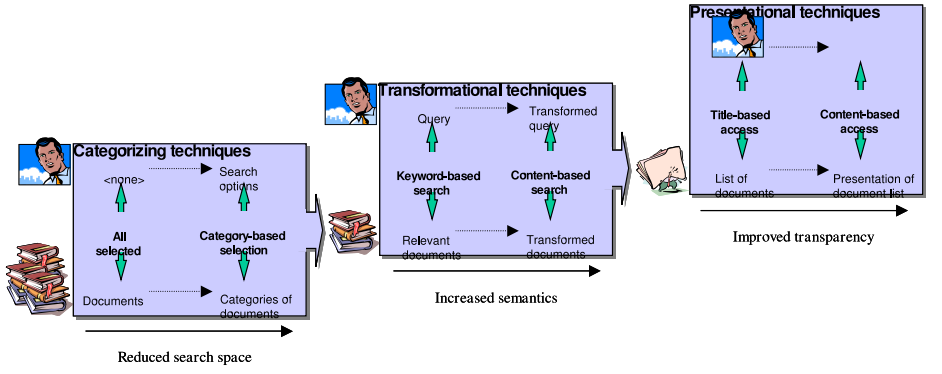


Figure 1. Linguistic techniques in FAST

Section 2 briefly presents the rationale of these linguistic techniques, we explain how these techniques were realized FAST in Section 3. The FAST search engine is discussed in Section 4 and also compared to some other large-scale search engines. Conclusions are found in Section 6.

2 Why Linguistics in Search Engines

A fundamental issue in information retrieval is the relationship between queries and documents. From a linguistic perspective, it is useful to abstract away from the exact words and focus on the information need expressed in the query and the semantic content of the documents. The document's relevance with respect to a query is not necessarily decided on the basis of words common to both query and document. The document is relevant to the extent that its content satisfies the user's need for information.

Linguistic techniques address the relevance of documents by exposing the underlying content or semantics of documents and queries. From a search perspective, the techniques fall into the following three categories (see Figure Figure 1):

- Categorizing techniques.
- Transformational techniques.
- Presentational techniques.

Each category is briefly explained below and illustrated by some common techniques.

One line of thought in linguistically motivated information retrieval is to restrict the search space to only those documents that may be relevant to the search context. This could be documents of a certain language, documents on a certain topic, or documents of a certain quality. All this requires linguistic analysis of documents before they are added to the search index. When posting a query, the user may increase the precision of the results by restricting the search space along these lines. In some cases it is appropriate that the search engine includes these restrictions automatically, as when offensive documents are filtered out of the result set. In other cases the user may want to select manually which categories should be considered in the search process. Using the Scirus.com library of scientific documents, for example, the user may restrict his search to a selection of 21 scientific categories. This search space reduction is achieved with a set of techniques that allow us to categorize documents along any number of dimensions [12].

Another line of thought is to transform queries or documents to representations that better facilitate document retrieval. Document indexes and queries need to be optimized or normalized with respect to each other. Assuming that the two queries car and cars denote the same information need, the search engine needs to treat documents with only car, documents with only cars and documents with both car and cars as equally relevant to both queries. Dealing with this inflectional variation is the task of lemmatization and stemming techniques. Lemmatization and stemming are part of the normalization task, which also includes strategies for handling transliteration and spelling variation. A related problem is the interpretation of phrasal queries. Even if the two queries Hotel Paris and Paris Hotel contain the same words, they may express two rather different intentions. The first query may request documents about the hotel called Hotel Paris, whereas the other may request information about hotels in Paris in general. To ensure that multiple-term queries are correctly interpreted, we need to recognize relevant phrases and make sure that the search engine takes this information into account. There are also phrases in queries that are more disturbing than useful. In the query give me information about search engines, the search would probably be improved if everything except search engines were deleted from the query. A necessary supplement to the transformational techniques above is spell-checking, which enables approximate search for misspelled search terms. In general, transformational techniques expose semantic aspects of queries and documents that should be taken into account in the search process.

The last group of linguistic techniques comes into play when documents are presented to the user. Since document titles and document names rarely

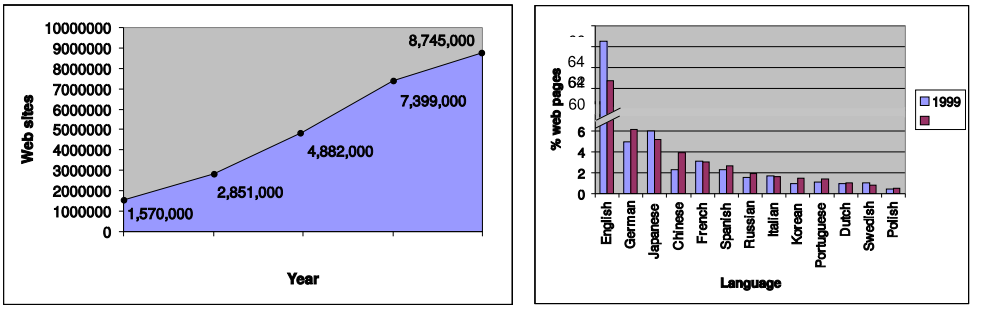


Figure 2. a) Number of web sites on the Internet[13] b) Distribution of languages on Internet[8]

give a good impression of the document content, it is often frustrating to find the right document among all these documents listed on the result page. Generating small document summaries that are shown on the result page is one way of telling the user what these documents are about (see for example [14]). Another technique, clustering, rearranges the result set according to pre-defined or dynamically generated categories[17]. In this way the user can choose the relevant categories after the search has been carried out and browse from one category to another. These presentational techniques improve the transparency of documents and speeds up the process of selecting the desired document from the result set.

3 Introducing Linguistic Components in FAST

For a large-scale global search engine, there are several concerns that need to be addressed. Due to the sheer size of the web, there are strict restrictions on memory consumption and response time. As seen from Figure Figure 2(a), the number of web sites is growing rapidly.Dealing with an increasing number of continuously changing documents, the search system must be effective enough to allow frequent indexing and short response times and efficient enough to run on the hardware available.

However, some of the most serious relevance problems in web search can only be solved by data-intensive linguistic components. Although English still accounts for more than 60% of the web pages, the share of non-English pages is increasing and the web is turning into a multi-lingual information

highway (see Figure Figure 2(b)). Within a few years, the non-English pages will probably outnumber the English ones. Solving linguistic problems in web search, we need deep knowledge of a number of languages to deal with the particularities of all these web pages.

The FAST approach to these linguistic problems is to use dictionary-driven techniques based on finite-state technology. This allows us to take full advantage of large lexical resources, while maintaining an acceptable response time. Our linguistic algorithms are language-independent, and new languages are introduced by plugging in new dictionaries. While most components are initially developed for English, the other languages are gradually added as lexical resources are built up and customers request these languages. In the following, we go into the realization of our linguistic techniques in the FAST search engine.

3.1 Language Identification

The language identifier of a search engine allows the user to select the language of the documents to be returned. If no language is selected, documents are retrieved and ranked independently of their language. As shown in Figure Figure 5, there is a language pull-down menu on AllTheWeb that lets you select the desired language.

The FAST language identifier has a dictionary that contains the relevant words in all relevant encodings of all languages to be recognized. For languages with no clear word boundaries, bigram lists are used instead of the word dictionary. The identifier can process HTML and plain text documents and identifies their language and encoding. Only the first part of the documents, like the first 100 words, is normally used in the identification process. Different strategies are used for different types of encoding-language pairs:

1. Languages with clear word boundaries (e.g. European languages): identify language and encoding by means of lookup in the dictionary of words with their frequencies for all such languages.

2. Languages with no clear word boundaries (Chinese, Japanese, Korean and Thai): identify language by checking the document against a frequency list of byte-bigrams.

Since many web documents are very short, our dictionary also contains words with medium frequencies and low frequencies. Another addition is that we take the structure of HTML documents and the meta information into consideration. In some cases, me may also use domain names like .DE

for Germany to find the most probable language of a web document. This has to be done with caution, as some domain names are linked to several languages (like .CH for Switzerland), some pages are written in languages not linked to the corresponding domain name (like English pages on .DE sites), and some pages are on domains with no fixed languages (like .ORG, .COM, and .NET).

For the current language identifier, which supports 52 languages, both recall and precision are above 99% for documents of more than 20 words. However, there is a tremendous amount of documents with little or no text at all on Internet. About 95–96% of the documents crawled are successfully tagged for language during indexing.

3.2 Offensive Content Filtering

Many web pages today contain text and images that may offend or disturb certain user groups. A particular concern is the abundance of pornographic material that tends to clutter the result sets. FAST has an Offensive Content Filter (OCF) that helps the users filter out documents that are sexually offensive.

The OCF component employs standard text categorization techniques and relies on a large dictionary of offensive words and phrases. These dictionary entries are generated from large collections of offensive web pages and are associated with weights that specify their influence in the filtering process. The filtering strategy itself is as follows:

1. Keep multilingual OCF dictionary of weighted words and phrases available in finite-state automaton during indexing.

2. Traverse first part of document and calculate score for offensive material based on dictionary lookup

Also offensive documents are indexed and can later be retrieved by the users. The documents are tagged as offensive and will not show up if the FAST offensive content filter has been activated on the front-end. If the filter is not activated, the documents will be retrieved just like any other documents on the web.

The current filter works for pornographic content in English, German, Italian, Spanish, and French. As many pornographic documents tend to contain keywords from several languages, the filter works reasonably well also for some other languages. The dictionary entries for each language span from a few hundred to several thousand.

A fundamental problem with this approach is the identification of offensive documents with little text. For documents that contain mostly images, text categorization methods can only be a partial solution. A combination of image recognition and text categorization may give better results, though this has to be investigated further.

3.3 Text Categorization

Words and phrases are colored by the domain in which they are used and should be interpreted within the context of this domain. A model in software engineering is something quite different from a model in fashion shows. For enterprise search solutions, it may make sense to group documents into separate sub-domains, or categories. Selecting the relevant categories, the user can make sure that the query is interpreted in the right context and only topically interesting documents are considered. Text categorization is the task of defining such categories in a domain and assigning documents to them. Although most categories are semantically motivated, it is also possible to categorize on the basis of document type, meta information, usage, etc.

Text categorization requires extensive analysis of the documents to be categorized and indexed. For each category, a dictionary defining the characteristic words and phrases needs to be set up. Weight information for each entry in this dictionary decides how these words and phrases together allow us to classify documents correctly. In some cases, structural information from the web documents may also be included in the categorization task. For example, HTML tags highlighting phrases or defining section titles may indicate which phrases are the prominent ones. Also, particular words like "homepage" or "Abstract" may in combination with other clues indicate that we are dealing with a homepage and an article, respectively.

The FAST text categorization module concentrates on content categorization and type categorization. Whereas content categorization identifies subject areas like Computer Science and Astronomy, the type categorization part can tell us whether the document is for example a homepage, an article or an abstract. The documents are categorized during indexing according to the following scheme:

1. Keep dictionary of words and phrases with associated categories (subject areas) available as finite-state automaton during indexing.

2. Content categorization: Calculate score for each category by looking up words and phrases in the dictionary.

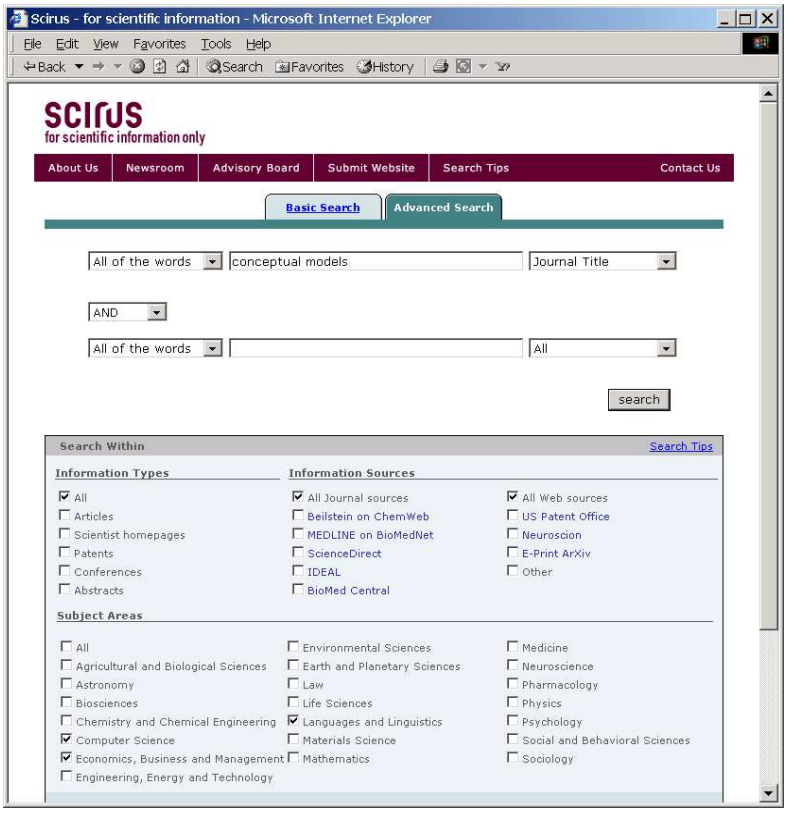


Figure 3. Scirus library of scientific information (www.scirus.com)

3. Extract meta information from structural information in document.

4. Type categorization: Deduce document type using rules that analyze the document's meta information.

Text categorization is useful in structured domains, in which separate subject areas can be defined by characteristic words and phrases. An example is the Scirus.com library of scientific documents shown in Figure Figure 3. While posting the query, the user may choose to restrict the search to a selection of the 21 scientific categories defined. In this particular case, the user is looking for papers on conceptual models within the categories Computer Sciences, Economics, Business and Management, and Languages and Linguistics. The system, which makes use of FAST's search technology, currently has an index of about 69 million science-related web pages.

For a general web search engine like AllTheWeb, text categorization does

not show the same potential. The Internet does not constitute a structured and well understood domain, and web pages are often of poor quality. Still, it may be possible to define categories for particular aspects of web pages, and we are now in fact looking into geographical categorization of documents on Internet.

3.4 Normalization

Even though there may be standardized keywords added to documents, most of the documents are just paragraphs of text in some language. Words are inflected according to the rules of the language, and there is great variation in word inflections, word selection and grammatical constructs. In many languages, words may also be spelled or transcribed in various ways. Unless we are able to treat inflections or spelling variants of a word as denotations of the same word, successful retrieval of a document requires that there is an exact match of the query terms and the words found in the document. Clearly, many relevant documents are missed if this is required, and we need ways of retrieving documents that contain other inflections or spellings of the same words.

The goal of normalization is to map different but semantically equivalent words and phrases onto one canonical representation. Without going into all kinds of normalization (see Arampatzis et al 2000 for an overview), we will here discuss the two types that are currently used in FAST: lemmatization and transliteration.

Lemmatization is the process of reducing an inflected word to its lemma, which is the abstract representation of the word independent of time, person, case, etc. Whereas the infinitive form is usually taken as the lemma form for verbs, the indefinite singular nominative form is used for nouns. Adjectives and adverbs are often reduced to their absolute form in the indefinite nominative masculine context. The following are some examples of lemmas generated from inflected words:

writes	→	*write*	*cars*	→	*car*
written	→	*write*	*better*	→	*good*
was	→	*be*	*best*	→	*good*

This lemmatization strategy is somewhat problematic in web search. If we replace all words in documents and queries with their lemmas, we increase recall by matching lemmas instead of inflected forms:

	Actual text	Lemmatized text
Query:	Good cars	good car
Document:	The best car	The good car

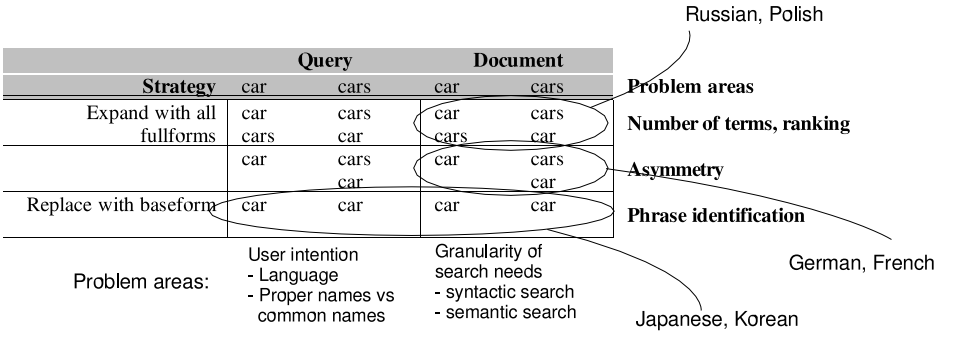


Figure 4. Lemmatization strategies

However, this prevents us from being able to search for exact phrases in documents and the query language must be known. A more conservative approach is just to add the lemmas (base forms) to documents and queries. This introduces some asymmetry, as the base forms are only added for the words that are not originally in lemma-compatible forms. A third strategy is to expand the document or the query with all inflected forms. The full form document expansion enables phrase search and can be done without knowing the language of the query, but increases the index size substantially and may lead to unexpected ranking results. Full query expansion is an alternative, if the query language is known, though also here there may be undesired ranking effects.

In general, for a multi-lingual search engine like AllTheWeb, where the query language is not always known, it is difficult to apply lemmatization on queries. Due to the lack of context, query lemmatization may in languages like German also lead to misinterpretation of common nouns as proper nouns and vice versa. Full-form document expansion is for obvious reasons only feasible for languages that are not too frequent on the web. For popular languages like German and French (see frequencies in Figure Figure 2(b)), baseform document expansion tends to be the most realistic strategy. Our lemmatization strategy for German, French, Japanese, Korean, Russian and Polish then looks as follows (see Figure Figure 4):

1. Identify language of document.
2. Keep lemmatization dictionary for identified language available as finite-state automaton during indexing.

3. For German and English: Expand document with base forms for adjectives and nouns.

4. For Polish and Russian: Expand document with all inflected forms for adjectives, verbs, common nouns, and proper nouns.

5. For Korean and Japanese: Replace words with base forms for verbs and adjectives in Japanese, and for verbs, adjectives and adverbs in Korean.

Lemmatization requires that large full form dictionaries be available during indexing. Our dictionary for German, for example, contains 1,016,492 full forms. For highly inflected languages, like Polish and Russian, we also employ morphological models that infer the correct inflections of proper names that are not in the dictionary. Japanese and Korean are special cases, as lemmatization and tokenization are done in the same step.

As seen from the strategies above, we add all inflected forms to the document for Russian and Polish. In practice this means that the indexes for these two languages increase by 600–800%. The moderate lemmatization strategy for German gives an index increase of only 5% and is therefore more attractive for languages that are very frequent on the web.

Transliteration of special characters is another issue in many languages. Take for example the word vnements (events) in French. Not only is it possible to spell this word in different ways, it is also quite common to misspell it and use the wrong accents or no accents at all. On AllTheWeb, there are 420,000 documents with vnements (old orthography), 35,000 documents with evenements (unaccented), 95,000 documents with evnements (half accented), 22,000 documents with evnements (half accented), 9,000 documents with venements (half accented), and 76,000 documents with the correct vnements. For the search engine to be able to retrieve all these documents, we need to normalize the spelling of words like vnements and map then onto the same words in the index.

The FAST strategy with respect to transliteration is to map accented characters onto unaccented characters where this does not impose any serious ambiguities. This is then done on both the document side and the query side. The word vnements above is mapped onto evenements, as these two accents in French does not lead to any change of semantics. Other accents or special characters, like the umlaut (, ,) in German, discriminate between semantically different words and are kept as they are in the documents and the queries.

3.5 Phrasing and Anti-Phrasing

The query' "mutual information" (explicit marking of phrase) gives us about 6,300 documents, and the top three documents are all about mutual information. Without explicit phrasing, more than a million documents are returned (AllTheWeb, March 2002). Since both mutual and information are used in so many other contexts, we get far more documents and none of the top ten documents are actually about mutual information. Some search engines uses proximity in general to rank documents where the terms are close to eachother higher than documents that have the terms further apart. This approach is general, but not all multi-term queries have a nature for proximity. Phrasing is important to restrict the number of documents found and to boost the search in the right direction. It may be described as the process of identifying and quoting consecutive terms that should be interpreted as a whole rather than as independent search terms.

Before phrasing queries, the search engine needs to build up a phrase dictionary from query logs and document collections. This list needs to be updated as new phrases emerge or others are not used any more. It may contain scientific expressions like the one above, but also person names, geographical names, book and song titles, names of organizations, slogans, etc. Since the Internet is not restricted to certain topics and new documents are continuously added, there are in principle an indefinite number of possible phrases. Another issue is whether all phrases should be treated in the same manner. Some phrases, like Albert Einstein, should always be interpreted as a whole and are often referred to as hard phrases. For soft phrases like data modeling, the individual terms may also be used independently of each other with the same semantic content.

The phrasing strategy adopted in FAST looks as follows:

1. Keep phrase list available in finite state automaton during querying.
2. Identify and quote longest left-most phrases in query from list of phrases.
3. Add bigrams of all consecutive query terms that are not part of a phrase as optional phrases.
4. Send the transformed query to the search engine, but offer the original query as an alternative on the result page.

For a query like New York Times sports, both New York and New York Times are recognized as phrases. The longest phrase is preferred, and the

query "New York Times" sports is consequently sent to the search engine. For a query like

New York art museum

the phrase New York is found in the phrase list. A bigram for the last two terms are introduced as an optional phrase. The final query is written in the ANY mode, which means that terms that need to be in the retrieved documents are preceded by a +:

+"New York" +art +museum "art museum"

With this strategy, we make a distinction between hard phrases and potential soft phrases in the query. Whereas the hard phrases are assumed to be in the phrase list, soft phrasing is achieved by adding these bigrams at the end of the query. An editorial board is then responsible for all updates of the phrase list, and new lists may be put into production every time a new index is generated (every 9-11 days).

In a similar vein, there are many queries that contain phrases that only disturb the search. Posting a query like where can I find The Economist, for example, the homepage of the magazine The Economist will not be found at all. If we remove the first part of the query, where can I find, The Economist shows up on top of the result list. The first part of the query prevents us from retrieving the correct document, as it is interpreted as equally important as The Economist by the search engine. These irrelevant phrases in the beginning of the queries are referred to as anti-phrases and should be removed before the documents are retrieved. Maintaining an extensive list of common anti-phrases, FAST handles these queries as follows:

1. Keep anti-phrase list available in finite state automaton during querying.
2. Phrase query.
3. Identify and remove longest left-most anti-phrase starting at position 1 in the query.

Consider the query

Where do I find New York?

The phrase New York is first identified from the phrase list. After the anti-phrase Where do I find has been recognized and removed, the following query is sent to the search engine:

"New York"

This strategy ensures that anti-phrasing does not remove parts of phrases. If phrasing had not been done prior to anti-phrasing, the query Give me information about The The would have been reduced to The rather than' "The

The" (both Give me information about the and Give me information about are in the anti-phrase list). It is worth noting that most anti-phrases start with a question word. However, we must be careful about which phrases to remove, as some of these question-like queries are used in titles of documents, like for example How to Tie a Tie.

Currently, we have several million phrases and thousands of anti-phrases available in English and German. Preliminary tests show that about 6% of the queries contain phrases, but only 0.2% contain anti-phrases.

3.6 Clustering

Traditional ranking of documents according to their relevancy to the query produces a flat list of ranked results. If the list is long, having an additional representation of the same result set in the form of a hierarchical tree instead of a flat list is a useful way to view the same set of search results. The tree nodes define groups of documents that have similar content, and give an immediate bird's-eye view of the topical distribution within the set of search results. Tree nodes may or may not have child nodes, i.e., subtopics within a topic can also be detected. The process of generating these hierarchical presentations of result sets is referred to as clustering.

The groups of documents with similar content effectively form a table-of-contents of the result set. Related to this grouping process, we may also obtain sets of concepts that are related to each group or to the result set as whole. For example, for a query like "image compression", we might detect that the phrases "fractal image compression" and "wavelet techniques" are descriptive of certain subsets of documents. This automatic grouping enables the user to quickly zoom in on subsets of documents that he finds interesting, while the sets of related concepts or phrases may provide ideas for queries that aid the user in obtaining, e.g., a more focused result set.

The FAST server is capable of doing both supervised and unsupervised learning, i.e., both classification and clustering. Some of the documents in the result set may be known to belong to some taxonomy, and, if configured to do so, the other documents in the result set can be attempted mapped on-the-fly into this taxonomy. For those documents that do not fit well into this hierarchy, an algorithm can be applied that tries to automatically detect topical clusters and devises descriptive names of these clusters. The clustering strategy is as follows:

1. Retrieve x highest ranked documents for the query posted.
2. For each document:

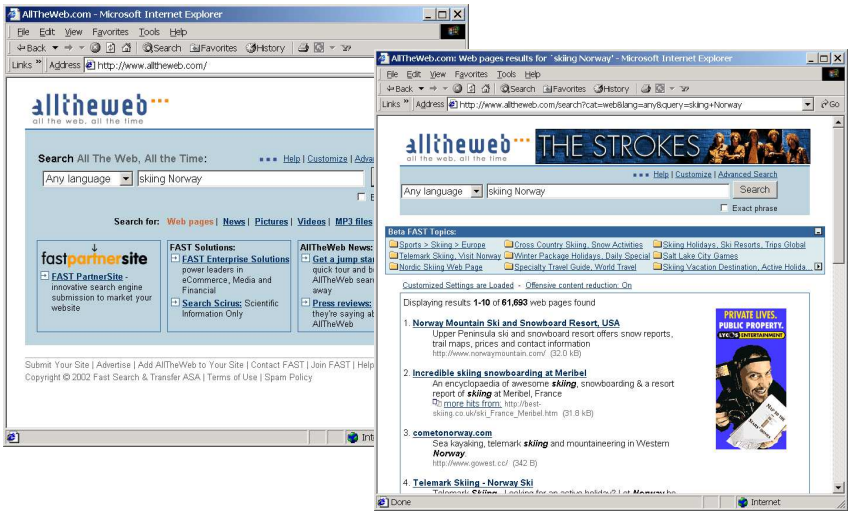


Figure 5. AllTheWeb search page and result page

(a) Extract words and phrases and assign a numeric measure of importance to each of them.

(b) Construct document vector (or use precomputed vector).

(c) For clusters already known: Try to map document to existing clusters.

(d) If mapping to existing clusters fail: Construct new cluster on the fly and assign document to new cluster

he unsupervised classification is done without any special knowledge of the semantics of words or phrases. Rather, the whole process is completely syntactic and query-driven, and from the viewpoint of the server, the example word "dog" is just a three-letter sequence without any special meaning. This means that the approach is largely language-independent, although having identified the language will help. (For example, if we know which language a document is in, we may exploit this in the vectorization process to skip certain very frequent words that carry little or no information. Example of such words in English are "the", "a", "to", etc.).

A beta version of the FAST module for on-the-fly classification and clustering of search results is deployed on AllTheWeb (see Figure Figure 5), where it is configured to make use of a cleansed version of the taxonomy from the Open Directory Project. The beta version works best for English documents. Note that for performance reasons on a large web search engine

with hundreds of queries per second, only parts of the documents are used for the classification.

Exact performance statistics for the FAST clustering server vary according to available hardware and how the server is configured, but for certain configurations over 100 queries/second may be processed on a typical PC. The time to vectorize 100 document summaries and cluster these may lie somewhere in the range of 5-30 milliseconds, depending on the chosen configuration.

4 Discussion

The importance of returning relevant documents on top of the result list has been highlighted by recent studies. As pointed out in Spink et al [16], the average user views 2.2 pages on the result page of AllTheWeb. At the same time, the average query length is only 2.4 terms and the total query session typically consists of about 8 queries. Only about 10% of the users care to use the additional search features provided by the Advanced Search mode. There is not much information given to the search engine, yet the search engine must be able to pick out the right documents among several billion documents and put them right on the top of the result page. Internal studies at FAST show that as much as 48% of the documents viewed by the users are among the top three documents on the result page. Incorporating techniques that increase the likelihood of finding relevant documents and make sure that they are presented on top of the result page in an understandable way is thus a top priority of any web search engine.

The linguistic techniques presented here serve different purposes, but also complement each other in important ways. Whereas normalization and anti-phrasing tend to increase the number of documents returned, language identification, phrasing, offensive content filtering and text categorization make the result set smaller and more focused. As shown in [9], many of these techniques can be combined to form a more specialized web context. For almost all techniques, there is a need to augment the standard approach with Internet-specific features to take full advantage of their potential. A thorough evaluation of each individual technique on Internet is very difficult, though we are now looking into test procedures in line with what has been presented in [3] and [5].

A rather delicate issue in web search is the balance between linguistic sophistication and computational feasibility. As shown in this paper, the techniques need to be adapted to the limitations of the rest of the search

engine. This does not only apply to the speed and memory requirements discussed above. Equally important is the way linguistic techniques interact with ranking algorithms and storing strategies.

The FAST search engine today has an index of about 625 million text pages, about 115 million multimedia pictures and videos, and more than 2 million MP3 songs. The index is refreshed every 9-11 days, which is rather extraordinary for an index of this size. In addition, there are more than 2,000 online news sources that are continuously crawled and updated. As of today, AllTheWeb has about 7-8 million queries per day and about 150,000 unique visitors every week. Adding the queries coming from partner portals like Lycos and T-Online, the FAST search engine has a total of about 30 million queries per day.

Other large search engines also employ linguistic techniques. Alta Vista (www.altavista.com), for example, has a battery of techniques that include language identification, spell-checking, stemming, phrase recognition, and some thesaurus support. As in FAST, they have adopted an expansion strategy for European languages and a reduction strategy for Asian languages in their lemmatization component. Language identification, spell-checking, and text translation are incorporated in the Google search engine (www.google.com). Autonomy uses linguistic techniques like stemming and transliteration to infer more conceptual representations of document content (www.autonomy.com). Query transformation is the main strategy of Ask Jeeves (www.askjeeves.com), which concentrates on natural language queries. All these techniques are comparable to the techniques employed by FAST, though it is hard to evaluate individual techniques without being affected by other features of the search engine. In the relevance test carried out by eTesting Labs in January 2001, AllTheWeb was ranked as the second best among six large-scale search engines [4]. In [3] 20 search engines's ability to return the homepages for airline queries is evaluated. AllTheWeb comes out on top of the list, ahead of Google and Microsoft.

5 Conclusions

This paper presented the linguistic techniques used in FAST's search engine and explored some of the technological challenges of integrating these techniques in a large-scale search architecture. Most of the techniques are now in production, and our initial results are indeed promising. It has to be added, though, that several of the techniques had to be adapted or augmented by additional strategies to be efficient in an Internet context. FAST is also

looking into other linguistic techniques that complement the ones already in production.

A particular issue is web search is spam detection. A large number of web pages on Internet are to be considered spam and should not be indexed for later retrieval. Our spam detection software is still being developed and shows how linguistics may be combined with other statistical and computational methods. In the future, we expect linguistic techniques to be tightly integrated with other knowledge-based methods to further increase the semantic focus of our search engine.

References

[1] Ambroziak, J., and Woods, W. Natural language technology in precision content retrieval. In *Proceedings of the International Conference on Natural Language Processing and INdustrial Applications (NLP+IA'98)* (1998).

[2] Cole, R., Mariani, J., Uszkoreit, H., Zaenen, A., and Zue, V. Survey of the state of the art in human language technology, 1995.

[3] Craswell, N., Hawking, D., and Griffiths, K. Which search engine is best at finding airline site home pages?

[4] eTesting Labs. Fast Search and Transfer, Inc. web search engine evaluation. Tech. rep., 2001.

[5] Hawking, D., Craswell, N., Bailey, P., and Griffihs, K. Measuring search engine quality. *Information Retrieval 4*, 1 (2001), 33–59.

[6] Kobayashi, M., and Takeda, K. Information retrieval on the web. *ACM Computing Surveys 32*, 2 (2000), 144–173.

[7] Kontzer, T. In search of business data. *InformationWeek.com* (January 2002).

[8] Langer, S. Natural languages and the World Wide Web.

[9] Lawrence, S. Context in web search. *IEEE Data Engineering Bulletin 23*, 3 (2000), 25–32.

[10] Lewis, D. D., and Jones, K. S. Natural language processing for information retrieval. *Communications of the ACM 39*, 1 (1996), 92–101.

[11] MAGAZINE, E. E-entertainment survey, October 2000.

[12] MANNING, C., AND SCHTZE, H. *Foundations of Statistical Natural Language Processing.* MIT Press, 1999.

[13] ONLINE COMPUTER LIBRARY CENTER. Size and growth. wcp.oclc.org/wc/stats/size.html.

[14] RADEV, D. R., AND MCKEOWN, K. Generating natural language summaries from multiple on-line sources. *Computational Linguistics 24*, 3 (1998), 469–500.

[15] RISVIK, K. M., AND MICHELSEN, R. Search engines and web dynamics. *Computer Networks (Amsterdam, Netherlands: 1999) 39*, 3 (2002), 289–302.

[16] SPINK, A., OZMUTLU, H., OZMUTLU, S., AND JANSEN, B. U.s. versus european web searching trends. In *Proceedings of SIGIR-02, 25th ACM International Conference on Research and Development in Information Retrieval* (2002).

[17] STRZALKOWSKI, T. *Natural Language Information Retrieval.* Kluwer Academic Publishers, 1999.

[18] SULLIVAN, D. Avoiding the search gap.