

UNIVERSITÀ DEGLI STUDI DI PISA
DIPARTIMENTO DI INFORMATICA
DOTTORATO DI RICERCA IN INFORMATICA

PH.D. THESIS: TD 5/04

High Performance Issues in Web Search Engines: Algorithms and Techniques.

Fabrizio Silvestri

SUPERVISOR
dr. Raffaele Perego

REFEREE
Prof. Guy E. Blelloch

REFEREE
Dr. Ronny Lempel

May 2004

Addr: Via F. Buonarroti 2, 56127 Pisa Italy
Tel: +39-050-315 3011 Fax: +39-050-313 8091 E-mail: fasilves@di.unipi.it
Web: <http://hpc.isti.cnr.it/~silvestr>

Abstract

For hundreds of years the mankind has organized information in order to make it more accessible to the others. The last media born to globally provide information is the Internet. With the Web, in particular, the name of the Internet has spread all over the World. Due to its impressive size and its high dinamicity, when we need to search for information on the Web, usually we begin by querying a Web Search Engine. A Web Search Engine maintains and catalogs the content of Web pages in order to make them easier to find and browse. Even though the various Search Engines are similar, each one of them differentiates from the other by the methods for scouring, storing, and retrieving information from the Web. Usually Search Engines search through Web pages for specified keywords. In response they return a list containing those documents containing the specified keywords. This list is sorted by a relevance criteria which try to put at the very first positions the documents that best match the user's query. The usefulness of a search engine to most people, in fact, is based on the relevance of results it gives back. This thesis tries to address some issues regarding some of the major challenges faced by Search Engines. In particular, since the size of the Web is rapidly growing, the main issues regard high performance algorithms for information management. Furthermore, nowadays Web Search Engines receive more than 200 million searches per day over a collection of several billion web pages indexed. This figures, in particular, can easily explain why in such environments the efficiency, as the effectiveness, of Search and Index algorithms have become issues. For these reasons in this thesis we are going toward proposing three novel techniques aimed at enhancing the performance of a Web Search Engine from three different angles. We studied a novel *Caching* policy which obtain high hit-ratio values: the aim of caching is to enhance the throughput of a Web Search Engine (i.e. the number of answered queries per second). We developed and tested a novel parallel *Indexing* architecture aimed at enhancing the number of documents indexed per hours. We studied a novel problem which is related to *Compression* of Web Search Engine indexes: the more compressed an index, the better the memory hierarchy of a system is exploited. We will show our main results in these areas and, whenever it is possible, we will compare them with the state of the art of each research area.

Acknowledgments

First of all, I want to express my gratitude to dr. **Raffaele Perego**, who allowed me to undertake my Ph.D. research in the ISTI (Information Science and Technology Institute) of the Italian National Research Council (CNR). He offered not only writing and academic guidance, but also general support and encouragement and, more important, he has been helping me to stay on the correct path towards my dissertation since my very first day as a Ph.D. student. I am also indebted to Prof. **Salvatore Orlando**, for his continuous support and fruitful discussions. He gave me a lot of useful ideas, many of which are contained within this work. He is also acknowledged for the critical reading of the manuscript. I am, also, very grateful to many other coworkers in the High Performance Laboratory: **Domenico Laforenza, Ranieri Baraglia, Paolo Palmerini, Nicola Tonellotto, Alessandro Paccosi, Antonio Panciatici**, and, last (but not least), **Diego Puppini**. A particular thanks goes to **Tiziano Fagni**, he is a long term friend as well as a loyal coworker. I want to express my deep gratitude to Prof. **Guy Blelloch** and to **Dan Blandford** for providing me with the source code of the implementation of their reordering algorithm. Dr. **Ronny Lempel**, along with Compaq Altavista's people, is acknowledged for providing me the *Altavista* query log. In particular, I would like to thank dr. Lempel for proving me the original implementation of its caching policy, *PDC*, and for the many helpful discussion we had during his (too) brief stay here in Pisa. Many thanks go to dr. **Antonio Gullì** and the company **Ideare S.p.A.** which, especially in the first part of my Ph.D., have supported my work providing me the query logs from the Tiscali Search Engine. I would like to express my gratitude also to Prof. **Giuseppe Attardi**, and to Prof. **Paolo Ferragina**, they have been part of my internal thesis committee and they gave me many helpful suggestions about the most interesting directions to follow. Now let me switch to Italian to ... ringraziare i miei familiari. Grazie **Antonietta, Lidia e Ezio**, sia per l'affetto che per il supporto che mi avete dato durante tutti questi anni trascorsi da studente.

[...] La quale opera io non ho ornata né ripiena di clausule ample, o di parole ampullose e magnifiche, o di qualunque altro lenocinio o ornamento estrinseco con li quali molti sogliono le loro cose descrivere et ornare; perchè io ho voluto, o che veruna cosa la onori, o che solamente la varietà della materia e la gravità del subietto la facci grata. [...]

Niccolò Machiavelli. Il Principe. 1513

[...] I have not set off this little work with pompous phrases, nor filled it with high-sounding and magnificent words, nor with any other allurements or extrinsic embellishments with which many are wont to write and adorn their works; for I wished that mine should derive credit only from the truth of the matter, and that the importance of the subject should make it acceptable. [...]

Niccolò Machiavelli. The Prince. 1513

Chapter 1

introduction

1.1 Motivations

Due to the explosion of the number of documents published on the Web, Web Search Engines (WSEs) have become the main mean of initiating the interaction with the Internet. Largest WSEs index today thousands of millions of multilingual web pages containing millions of distinct terms. Due to the peculiarities and the huge size of the Web repository, traditional Information Retrieval (IR) systems developed for searching smaller collections of structured or unstructured documents appear inappropriate for granting retrieval effectiveness on the Web. Most components of an IR system must be rethought in order to address the problem of effectively and efficiently searching and retrieving information from the Web. Consider, as an example, the problem of deciding which documents are relevant and which are not with respect to a user query. This hard task is commonly delegated to a ranking algorithm which attempts to establish a ordering among the documents retrieved. Documents appearing at the top of this ordering are the ones considered to be more relevant. The two most accepted metrics to measure ranking effectiveness are: *Precision* (i.e. number of relevant documents retrieved over the total number of retrieved documents) and *Recall* (i.e. number of relevant documents retrieved over the total number of relevant documents in the collection) [148]. In traditional IR, we can assume that the documents in the collection originate from a reputable source and all words found in a document were intended for the reader. Ranking can simply be based on statistics performed over word frequencies. The same assumption does not hold on the Web where content is authored by sources of varying quality and words are often added indiscriminately to boost the page's ranking [11, 13, 58]. Moreover, as the size of the indexed collection grows, since users usually only look at the first few tens of results, a very high *precision* has to be preferred even at the expense of the *recall* parameter.

Similar considerations can be done for others key components of an IR system. The size of the Web data repository along with its exponential growth, the heterogeneity and dynamicity of web data, are all challenging problems justifying the structural complexity of the software architecture of modern WSEs that exploit a variety of novel technologies

developed in several related research areas such as databases, parallel computing, artificial intelligence, statistics, etc.

In this thesis we are going to illustrate some novel results regarding three different WSE issues: *caching of search engine query results*, *indexing*, and *index compression*.

1.2 Contribution of the thesis

Caching search engine query results

This chapter discusses the design and implementation of an efficient two-level caching system aimed to exploit the locality present in the stream of queries submitted to a Web search engine. Previous works showed that there is a significative temporal locality in the queries, and demonstrated that caching query results is a viable strategy to increase search engine throughput. We enhance previous proposals in several directions. First we propose the adoption of a new caching strategy. We called this new policy SDC (Static and Dynamic Caching). Basically, it consists of storing the results of the most frequently submitted queries in a *fixed-size read-only* portion of the cache. The remaining portion of the cache is used by the queries that cannot be satisfied by the read-only one according to a single cache replacement policy. Moreover, we show that search engine query logs also exhibit spatial locality, since users often require subsequent pages of results for the same query. SDC also take advantage of this type of locality by exploiting a sort of *speculative* prefetching strategy. We experimentally demonstrate the superiority of our SDC over other policies. We measured the hit-ratio achieved on three large query logs by varying the size of the cache, the percentage of read-only entries, and the replacement policy used for managing the dynamic cache. Finally, we propose an implementation optimized for concurrent accesses, and we accurately evaluate its scalability.

Indexing Web documents

We introduce a novel architecture for a parallel indexer for Web documents. By exploiting both data and pipeline parallelism, our prototype indexer efficiently builds a partitioned and compressed inverted index, a suitable data structure commonly utilized by modern Web Search Engines. We discuss implementation issues and report the results of preliminary tests conducted on a SMP PC.

Index compression issues

Granting efficient accesses to the index is a key issue for the performance of Web Search Engines (WSE). In order to enhance memory utilization and favor fast query resolution, WSEs use *Inverted File* (IF) indexes where the posting lists are stored as sequences of *d_gaps* (i.e. differences among successive document identifiers) compressed using vari-

able length encoding methods. These encoding algorithms exploit a property of the posting lists of an IF called clustering.

We introduce a new problem called the *Assignment Problem* and we give a formal definition for it. Furthermore we present a series of algorithms aimed to find an assignment of the numerical identifiers to documents which minimize the average values of *d_gaps*. In order to validate the approach followed, we run several simulations over the Google contest collection.

The results of the tests show that our approach allows to obtain an IF index which is, depending on the *d_gap* encoding method chosen, up to 17.70% smaller than the one built over randomly assigned document identifiers. We will analyze both analytically and empirically the performance of our algorithms in terms of both space and time occupied.

1.3 Past Work

In a previous work we developed MOSE (My Own Search Engine) a *parallel and distributed WEB search engine*, and investigated the possibilities offered by parallel techniques in the Web IR field. In [113] we described the architecture of MOSE. MOSE was specifically designed to efficiently exploit affordable parallel architectures, such as a cluster of workstations. Its modular and scalable architecture can be easily adjusted to fulfill the bandwidth requirements of the application at hand. Both *task-parallel* and *data-parallel* approaches are exploited within MOSE in order to increase the throughput and efficiently use communication, storage and computational resources.

The IR core of MOSE is composed of the *Indexer* and the *Query Analyzer* (QA) modules. In the paper we only focused on the QA whose functionalities were carried out by two pools of parallel processes: *Query Brokers* (QBs) and *Local Searchers* (LSs). MOSE parallel and distributed implementation exploited a data-parallel technique known as *document partitioning*.

This is the outline of the tasks performed by MOSE: the spidering phase returns a set of subcollections of documents with similar sizes. The subcollections are then indexed independently and concurrently by parallel *Indexers*. The result of the indexing phase is a set of p different indexes containing references to disjoint sets of documents. The p indexes are then taken in charge by a data-parallel QA whose task is to resolve user queries on the whole collection. To this end the QA used k QBs and p LSs. The k QBs run on a front-end workstation, and fetch user queries from a shared message queue. Every fetched query is then broadcasted to the associated p LSs (*workers*), possibly running on different workstations. The p LSs satisfy the query on the l references to most relevant documents contained within each subcollection. The QB waits for all the $l \cdot p$ results and chooses among them the l documents with the highest ranks. Finally, such results were returned to the requesting user. In order to manage concurrently more queries and to better exploit LSs' bandwidth, k QBs are introduced within a QA. System performance can be further increased by replicating the QA in n copies. Appropriately choosing parameters n , k , and p we could obtain three different parallelization strategies:

- *Task parallel*: in which a processor farm structure is exploited;
- *Data parallel*: the input database is partitioned and each partition is managed by distinct LSs;
- *Task + Data parallel*: a combination of the above two strategies.

To perform experimental evaluations we used a collection of html documents as a benchmark and conducted preliminary experiments on a cluster of three SMP Linux PCs. The results we obtained highlighted the greater performance resulting from exploiting a hybrid *Task + Data* parallelization strategy over a pure *Task-parallel* one.

Some problems encountered during the design of MOSE still remain open. First of all we would like to extract performance models for all the three parallelization strategies. Such models may drive the selection of the most suitable model when system parameters such as target architecture or dataset contents are varied. We are also interested in investigating the advantages and disadvantages of using a *Document Partitioning* strategy against a *Term Partitioning* one and from this analysis we would like to draw theoretical models helping the choice of the most suitable distribution schema. Moreover, as we already said, during the realization of MOSE we faced directly with the development complexity of this kind of software so we think that some kind of software engineering is needed to facilitate reuse and maintenance of source code.

1.4 Outline

In this thesis we are going to proceed as follows. In Chapter 2 a survey of the current state of the art in the general, and wide, field of Web Information Retrieval is presented. Chapter 3 is devoted to the explanation of some novel results we obtained with a new caching policy for WSEs' results. In Chapter 4 we introduce and analyze a novel indexing architecture thought for indexing Web documents. In Chapter 5 we present some results about an interesting problem related to the indexing phase of a WSE: the *Assignment Problem*. Finally in the last chapter we present some conclusions and future work proposals.

Contents

1	introduction	3
1.1	Motivations	3
1.2	Contribution of the thesis	4
1.3	Past Work	5
1.4	Outline	6
2	State of the art of WSEs' technology	15
2.1	Sequential Techniques	15
2.1.1	Crawling	16
2.1.2	Indexing	18
2.1.3	Query Analysis	20
2.1.4	Ranking	21
2.2	Parallel and Distributed Techniques	22
2.2.1	Parallel Crawling	22
2.2.2	Parallel Indexing	24
2.2.3	Efficient Query Brokering	25
2.2.4	Peer To Peer	27
3	Caching Search Engine Query Results	31
3.1	Introduction	32
3.2	Related Work	34
3.3	Analysis of the query logs	36
3.3.1	Temporal locality	37
3.3.2	Spatial locality	38
3.3.3	Theoretical upper bounds on the cache hit-ratio	39
3.4	The SDC policy	40
3.4.1	Implementation Issues	41
3.5	Experiments with SDC	42
3.5.1	Experimental setup	42
3.5.2	SDC without prefetching	43
3.5.3	SDC and prefetching	44
3.5.4	Freshness of the Static Set	46
3.6	Concurrent cache manager	46

3.7	Conclusions and Future works	49
4	Indexing Web Documents	59
4.1	Indexing in Web Search Engines	60
4.2	A pipelined indexing system	61
4.2.1	Experimental results	65
4.3	Results assessment and future works	67
5	Index Compression Issues	69
5.1	Compression of Inverted File	70
5.1.1	Bitwise encoding.	71
5.1.2	Bytewise encoding.	73
5.2	The Assignment Problem	75
5.2.1	The <i>B&B</i> Algorithm	76
5.3	Collection Model	78
5.4	Our Algorithms	79
5.4.1	Top-down assignment	79
5.4.2	Bottom-up assignment	83
5.5	Experimental Setup	86
5.6	Comparisons	87
5.7	Summary	90
	Conclusions	91
	Bibliography	93

List of Figures

3.1	Architecture of a large-scale, distributed WSE hosting a query results cache.	33
3.2	(a) Number of submissions of the most popular queries contained in the three query logs (log-log scale). (b) Distances (in number of queries) between subsequent submissions of the same query.	50
3.3	Spatial locality in the three query logs analyzed: (a) percentage of queries as a function of the index of the page requested; (b) probability of the occurrence of a request for the i -th page given a previous request for page $(i - 1)$	51
3.4	Hit-ratios obtained on the three query logs by using different replacement policies as a function of the ratio between static and dynamic cache entries: (a) <i>Tiscali</i> , (b) <i>Excite</i> , (c) <i>Altavista</i> . The tests performed are referred to a cache whose size was fixed to 50,000 entries.	52
3.5	Hit-ratios obtained on the three query logs by using different replacement policies as a function of the ratio between static and dynamic cache entries: (a) <i>Tiscali</i> , (b) <i>Excite</i> , (c) <i>Altavista</i> . The tests performed are referred to a cache whose size was fixed to 256,000 entries.	53
3.6	Hit-ratios achieved on (a) <i>Tiscali</i> , (b) <i>Excite</i> , (c) <i>Altavista</i> logs as a function of the size of the cache and the f_{static} used.	54
3.7	Hit-ratios obtained on the three query logs by using different replacement policies as a function of the prefetching factor: (a) <i>Tiscali</i> , (b) <i>Excite</i> , (c) <i>Altavista</i> . The tests performed are referred to a cache whose size was fixed to 256,000 entries for <i>Tiscali</i> and <i>Altavista</i> ; 128,000 for <i>Excite</i>	55
3.8	Hit-ratio variations vs. prefetching factor on the Excite query log with a cache of 32,000 elements and <i>LRU</i>	56
3.9	Hit-ratios obtained on the three query logs by using different replacement policies as a function of the ratio between static and dynamic cache entries on the <i>Altavista</i> query log. The tests performed are referred to a cache whose size was fixed to 256,000 entries. The training set in this tests is composed by the first million queries submitted.	56
3.10	Scalability of our caching system for different values of f_{static} as a function of the number of concurrent threads used.	57

4.1	Construction of a distributed index based on the document partition paradigm, according to which each local inverted index only refers to a partition of the whole document collection.	61
4.2	Forms of parallelism exploited in the design of a generic <i>Indexer</i> , in particular of the first module (a) $Indexer_i^{pre}$, and the (b) $Indexer_i^{post}$ one. . .	63
4.3	Inverted file produced by the <i>Flusher</i>	64
5.1	The structure of an Inverted File in our "reduced" model.	71
5.2	Two examples of integer encoded following a variable-byte code.	73
5.3	The scalability of the <i>B&B</i> algorithm when varying the size of the collection reordered.	78
5.4	(a) Time (in seconds) and (b) Space (in KBytes) consumed by the proposed transactional assignment algorithms as a function of the collection size.	87

List of Tables

3.1	Main characteristics of the query logs used.	37
3.2	Hit-ratio upper bounds for each query log as a function of the prefetching factor.	40
4.1	A toy text collection (a), where each row corresponds to a distinct document, and (b), the corresponding inverted index, where the first column represents the lexicon, while the last column contains the inverted lists associated with the index terms.	62
4.2	Sequential execution times for different sizes of the document collection.	65
4.3	Total throughput (GB/s) when multiple instances of the pipelined <i>Indexer_i</i> are executed on the same 2-way multiprocessor.	66
5.1	The results obtained on the Google Collection by the <i>B&B</i> algorithm. The results are expressed as bit per posting after the encoding phase. Interp. coding is the Binary Interpolative coding, while Var. Byte is the Variable Byte technique.	76
5.2	Performance, as average number of bits used to represent each posting, as a function of the assignment algorithm used, of the collection size (no. of documents), and of the encoding algorithm adopted. The row labeled “Random assignment” reports the performance of the various encoding algorithms when DocIDs are randomly assigned.	89

List of Algorithms

1	$TDAssign(\tilde{D}, H)$: the generic <i>top-down</i> assignment algorithm.	81
2	The k -scan assignment algorithm.	84
3	The <i>select_members</i> procedure.	85

Chapter 2

State of the art of WSEs' technology

Abstract

In this chapter we present some research issues in the field of *Highly Responsive* Web IR systems, i.e. WSEs that provide both *High Performance*, intended for finding the best compromise among better exploitation of system resources (i.e. high system *throughput*) and low processing cost (i.e. low system *latency*) even at heavy load conditions (i.e. system *scalability*); and *Quality of Service* (QoS) requirements, intended as designing systems aware of *Timeliness*, and *High Availability* requirements.

Web search services have proliferated in the last years. Users have to deal with different formats for inputting queries, different results presentation formats, and, especially, differences in the quality of retrieved information [91]. Also performance (i.e. search and retrieval time plus communication delays) is a problem that has to be faced while developing such a type of application which may receive thousands of requests at the same time.

Most search engine developments is done within competitive companies which do not publish technical details.

In this chapter we review some of the most notable ongoing research projects in the Web IR area. In Section 2.1 we review the main results concerning the techniques for searching the Web. Next, in Section 2.2, we describe software architectures for either parallel, or distributed WSEs modules.

2.1 Sequential Techniques

Parallel and Distributed processing is an enabling technology for efficiently searching and retrieving information on the Web. Despite this fact, enhancements to sequential IR methods are very important.

The remainder of this section presents main innovations in WEB IR sequential algorithms and is organized as follows. In Subsection 2.1.1 we depict the state of the art in

crawling techniques. We continue discussing the ongoing research projects in the field of indexing and query analysis (Subsection 2.1.2 and 2.1.3) and, finally, in Subsection 2.1.4 we describe ranking strategies for IRSs for Web related data.

2.1.1 Crawling

Several papers investigate the design of effective crawlers for facing the information growth rate. There are several aspects aimed at improving classical spidering schemes [36, 109, 29, 116, 46, 3, 107, 101].

Main efforts are oriented toward finding an effective solution for reducing the number of sites visited by Crawlers. The main contributions are:

- *URL-ordering*;
- *Focused Crawling*; and
- *Incremental Crawling*.

The *URL-ordering* technique consists of sorting the list of URLs to be visited using some importance metrics and in crawling the Web according to the established ordering. This technique impact both the repository refresh time and the resulting index quality since the most important sites are chosen first.

In [36], Garcia-Molina *et al.* investigate three importance measures to establish site importance: *Similarity to a Driving Query Q*, where the importance is measured as the distance among the URLs content and a query *Q*, *Backlink Count* where the importance is the number of URLs linking to the current URL, and *PageRank* which is based on the PageRank ranking metrics [114, 19]. From the paper we could devise two main aspects. In spidering algorithms which consider only *PageRank* and *Backlink count*, the PageRank strategy outperforms the other due to its non-uniform traversing behavior: going in depth when the importance of the children is high enough, moving to the siblings whenever children nodes contain unimportant documents. On the other hand, when a similarity driven crawling algorithm is used the *PageRank* strategy is comparable to the a *Breadth First* traversal. This happens because when a page is authoritative with respect to a particular topic, its children are likely to have a high importance too. In their work the authors restricted the crawling space to the Stanford University Web pages. Thus, the resulting repository represents only a very small fraction of the entire Web space.

In contrast to the results obtained by Garcia-Molina *et al.*, the authors of [109] showed that the breadth-first search order discovers the highest quality pages during the early stages of the crawling process and, as crawling progresses, the quality of the downloaded pages deteriorates. They argue that the breadth-first strategy performs better due to the high probability of an important sites to be linked from another one. Thus, crawlers using breadth first, with high probability, will visit very soon a large number of important site. Moreover their system, consisting of a 667 MHz Compaq AlphaServer ES40 with 16 GB of RAM, took over a day to compute the PageRanks of a graph on 351 million pages,

despite the fact that they had the hardware resources to hold the entire graph in memory. Since using PageRank to steer a crawler would require multiple such computations over larger and larger graphs, they conclude that the PageRank solution is in practice infeasible for very large document repositories.

Focused Crawling is an argument very close to *Url-Ordering*. A focused crawler is designed to only gather documents on a specific topic, thus reducing downloads and the amount of network traffic.

Experiences with a focused crawler are presented in [29]. The crawler starts by using a canonical topic taxonomy and user specified starting points (e.g. bookmarks). A user marks interesting pages as he browses. Such links are then placed in a category in the taxonomy. This was bootstrapped by using the Yahoo hierarchy (260,000 documents).

The main components of the focused crawler are a classifier, a distiller and a crawler. The classifier makes relevance judgments on pages to decide on link expansion, and the distiller determines centrality of pages to determine visit priorities. The latter is based on connectivity analysis. In order to evaluate the proposal, authors consider the *harvest ratio*, i.e. the rate at which relevant pages are acquired, and how effectively irrelevant pages are filtered away.

They state that it is desirable to start from keyword-based and limited-radius search. Another observation was that the web graph is rapidly mixing: random links rapidly lead to random topics. At the same time, long paths and large subgraphs exist with topical coherence.

A type of focused crawling which concentrates on the "*hidden web*" is investigated in [116]. The objective is to gain access to content "hidden" behind web search forms. The researchers propose a task-specific, human-assisted approach, and built a system called HiWE (Hidden Web Exposer). They model a form as a set of (element, domain) pairs and try to determine suitable input values based on labels, form layout etc. The label value set (LVS) is populated using explicit initialization, built-in categories (dates), wrapped data sources and crawling experience. The crawler seems to perform better on larger forms (more descriptive labels and finite domains). This approach, however, could become rapidly obsolete since in the near future the majority of dynamic pages will be probably generated by *Web Services* [145] which expose their interface using an *ad-hoc* protocol (*WSDL* and *UDDI*) that make it publicly available to other applications [129].

The use of context graphs to guide a focused crawler is detailed in [46]. The authors state that a major problem in this area is the appropriate assignment of credits to pages during the crawling process. For example, some off-topic pages may lead to on-topic pages, as a result of page hierarchies. To deal with this problem, a context focused crawler was built which uses a general search engine to get the pages linking to a specific document and builds a "*context graph*" for the page. This graph is then used to train a set of classifiers which assign documents to different categories based on their expected link distance from the target. Graphs and classifiers are constructed for each seed document, and allow crawler to gain knowledge about topics that are directly or indirectly related to the target topic.

The context focused crawler was compared to an ordinary breadth-first crawler and a

"traditional" focused crawler. The metric used was the proportion of relevant downloaded pages. The experiments showed that the context focused crawler outperformed the other approaches.

A focused crawler which tries to learn the linkage structure while crawling is described in [3]. This involves looking for specific features in a page which makes it more likely that it links to a given topic. These features may include page content, URL structure, link structure (siblings etc.). The authors claim that the learning approach is a more general framework than assuming a standard pre-defined structure while crawling.

A *Web Topic Management System* (WTMS) is described in [107]. The authors use a focused crawling technique which only downloads pages which are "near" a relevant page (parent, children and siblings). The crawler will also consider all URLs containing the topic keyword. Finally, it keeps a relevance threshold for a particular area (directory), and will stop downloading from that location if relevance drops below a certain level. The other aspect of this work is how to visualize topics, using logical groupings of sites based on hubs and authorities [86].

Incremental Crawling concerns with the problem of the data repository freshness. One can choose among two different repository management strategies. The first consists of rebuilding the entire archive from the scratch the second consists of updating the changed important pages in the repository and replacing "less-important" pages with new and "more important" pages. The major difficulty with this approach resides in the estimation of the *freshness* of Web pages needed to reduce the number of *Needless Downloads*. There are some papers trying to address this last problem [34, 32, 18, 35, 49, 71].

In [18] the authors evaluate some simple and easy-to-incorporate modifications to web servers resulting in significant bandwidth savings. Specifically, they propose that web servers export meta-data archives describing their content. The meta-data should identify the recently changed pages without the crawler needing to download them and should indicate in advance the bandwidth needed to download modifications.

In their article they study different approaches to meta-data distribution tailored for even medium/small and large website. They evaluate the effects of the proposals gathering evolution data for 760,000 pages and several web server access logs. They ran simulated crawls using both the typical crawling strategy and a meta-data enabled crawling strategy. From their data and experiments they conclude that significant web server resources (e.g., bandwidth) can be saved if servers export meta-data.

In [71] the authors provide a solution which frees web search engines from the burden of repeatedly querying for Web server updates on a per-URL basis. They provide an algorithm for Web servers that can be used to batch the push of updates to the search engines or a middleware (if one exists).

2.1.2 Indexing

Various access methods have been developed to support efficient search and retrieval over text document collections. Inverted files have traditionally been the index structure of choice for the Web. Commercial search engines use custom network architectures and

high-performance hardware to achieve sub-second query response times using such inverted indexes.

When the collection is small and indexing is a rare activity, optimizing index-building is not as critical as optimizing run-time query processing and retrieval. However, with a Web-scale index, index build time also became a critical factor for two reasons: *Scale and growth rate*, and *Rate of change*.

A great deal of work has been done on inverted-index based information retrieval issues, including index compression [104, 110, 151], and incremental updates [37, 146, 151].

Moffat and Zobel [104] describe an inverted list implementation that supports jumping forward in the compressed list using skip pointers. This is useful for document based access into the list during conjunctive style processing. The purpose of these skip pointers is to provide synchronization points for decompression, allowing just the desired portions of the inverted list to be decompressed.

In the *skipped* structure proposed in the paper, each inverted list is divided into blocks each containing a fixed number of pointers. Codes (the *skips*) for the *critical document numbers* that start each block and for the bit-location in the compressed list of the next block are intertwined with the pointers themselves, so that the pointers in a block need not to be decoded if it is clear that the candidate sought cannot appear in that block. Within each block, each pointer is still *code-dependent* upon its predecessor, and so the search for a candidate takes place in two stages – first the skips are decoded sequentially until the block that must contain the candidate (if it appears at all) is determined, and then the pointers within that block are decoded.

In [110] the schema presented in the paper above is enhanced permitting an improved random access to postings lists. They adopt a “*list header*” approach [8], but compress the list of critical document number, and describe a different mechanism for coding – and then quickly decoding – the contents of each block.

Incremental updates of indexes have been studied extensively in the field of classical IR systems. In this kind of systems the considered update operations are insertion and deletion of documents from the collection¹.

The authors of [37] proposed a data structure to perform efficient on-line update of the index with low performance loss. The structure they use is organized as block of disk storage allocated for *index blocks*. Together, the index blocks make up the *index*, that combines the functions of the dictionary and postings file, thus the index contains both index terms and postings into a single file. Updates are buffered in main memory until they can be applied to disk. A background process continuously cycles through index storage applying updates and re-writing the index. Update throughput is thus a function of the size of the main memory buffer and the period of an update cycle.

Another approach to incremental updates is shown in [146]. The authors show a *dual structure* index strategy to address the problem of modifying index structures in place as documents arrive. The index is composed by two distinct pieces: short lists and long

¹Different from Web repositories where modifications to documents are permitted.

lists. Lists are initially stored in a short list data structure; as they grow they migrate to a long list data structure. In the same paper the authors present a family of disk allocation policies for long lists. Each policy dictates, among other things, where to find space for a growing list, whether to try to grow a list in place or to migrate all or parts of it, how much free space to leave at the end of a list, and how to partition a list across disks.

2.1.3 Query Analysis

Popular search engines receive millions of queries daily, a load never experienced before by any IR system. Additionally, search engines have to deal with a growing number of Web pages to discover, to index, and to retrieve. To compound the problem, search engine users want to experience small response times as well as precise and relevant results for their queries. In this scenario, the development of techniques to improve the performance and the scalability of the results becomes a fundamental topic of research in IR. One effective alternative for improving performance and scalability of information systems is caching. The effectiveness of caching strategies depends on some key aspects, such as the presence of reference locality in the access stream, and the frequency at which the database is being cached and updated.

The works studying caching techniques applied to WSEs, are very few [100, 31, 124].

The work in [100] considered the problem of caching search engine query results. The study uses a log containing one million queries submitted to the search engine Excite [52]. It proposes policies to cache query results that are based on the reference locality observed in the Excite log. However, their work did not include any implementation of the proposed caching policies.

In [31] the authors present a *semantic cache mechanism* designed for meta-searchers querying heterogeneous Web repositories. Semantic caching is based on the representation of cached data and remainder queries for fetching data from remote servers.

The authors of [124] described and evaluated the implementation of caching schemes that improve the scalability of search engines without altering their ranking characteristics. The work has been tested integrating a real WSE (TodoBR [54]) with three caching schemes. The first one implements a cache of query results, allowing the search engine to answer recently repeated queries at a very low cost, since it is not necessary to process those queries. The second one implements a cache of the inverted lists of query terms, thus improving the query processing time for the new queries that include at least one term whose list is cached. The third caching scheme combines the two previous approaches and it is called *two-level* cache. The experimental results described in the paper show that the two-level caching generally outperforms the others. The two-level cache allows increasing the maximum throughput (the number of queries processed per second) by a factor of three, relative to an implementation with no cache. Furthermore, the throughput of the two-level cache is up to 53% higher than the implementations using just cache of inverted lists and up to 36% higher than the cache of query results.

2.1.4 Ranking

Since users usually look only at the very first pages returned by a Web search engine [130], it is very important to effectively rank the results returned for the submitted queries.

The two main techniques used in ranking algorithms for Web pages are: *Statistical* (i.e. based on words occurrence in the pages), and *Link Based* (i.e. based on importance inferred from informations on the structure of the Web graph). In this document we focus our attention on link based techniques.

Kleinberg's seminal paper [86] on *hubs* and *authorities* introduced a natural paradigm for classifying and ranking web pages, setting off an avalanche of subsequent works [28, 5, 9, 14, 17, 25, 27, 38, 45, 48, 75]. Kleinberg's ideas were implemented in *HITS* as part of the *CLEVER* project [25, 28].

With the *CLEVER* project the authors oriented their efforts toward the development of a Search Engine which locates not only a set of relevant pages but also those relevant pages of the highest quality [28]. *CLEVER* is a Search Engine that analyzes hyperlinks to uncover two types of pages:

- authorities, which provide the best source of information on a given topic; and
- hubs, which provide collections of links to authorities.

The *HITS* (which stands for Hyperlink-Induced Topic Search) algorithm computes lists of hubs and authorities for Web search topics. Beginning with a search topic, specified by one or more query terms, the *HITS* algorithm applies two main steps:

- a sampling component, which constructs a focused collection of several thousand Web pages likely to be rich in relevant authorities; and
- a weight-propagation component, which determines numerical estimates of hub and authority weights by an iterative procedure.

The main problem with this method is in its first step where a sample collection focused on a particular topic is built. To perform this step they propose to submit a query to a search engine. This kind of approach, thus, cannot rank pages without the help of another search engine.

Around the same time, Brin and Page [19, 114] developed a highly successful search engine, Google [53], which orders search results according to *PageRank*, a measure of authority of the underlying page. The *PageRank* of a page is computed by weighting each hyperlink proportionally to the quality of the page containing the hyperlink. To determine the quality of a referring page, they use its *PageRank* recursively. *PageRank* can be formally defined to be the stationary distribution of the infinite random walk p_1, p_2, p_3, \dots where each p_i is a node in the graph G induced by the Web structure. Each node is equally likely to be the first node p_1 . To determine node p_{i+1} with $i > 0$ a biased coin is flipped: with probability ϵ node p_{i+1} is chosen uniformly at random from all nodes in G , with probability $1 - \epsilon$ node p_{i+1} is chosen uniformly at random from all nodes q such that edge (p_i, q) exists in G .

The PageRank is the dominant eigenvector of the probability transition matrix of this random walk. This implies that when PageRank is computed iteratively using the above equation, the computation will eventually converge under some weak assumptions on the values in the probability transition matrix.

The PageRank algorithm assigns a score to each document independent of a specific query. This has the advantage that the link analysis is performed once and the can be used to rank all subsequent queries.

In [92] the authors study a new approach for finding hubs and authorities, which they call *SALSA*, the Stochastic Approach for Link-Structure Analysis. *SALSA* examines random walks on two different Markov chains which are derived from the link-structure of the Web: the authority chain and the hub chain. The principal community of authorities (hubs) corresponds to the pages that are most frequently visited by the random walk defined by the authority (hub) Markov chain. *SALSA* and the Kleinberg's Mutual Reinforcement approach are both in the framework of the same meta-algorithm. The most interesting result is that the ranking produced by *SALSA* is equivalent to a weighted in/out degree ranking. This make *SALSA* computationally lighter than the Mutual Reinforcement approach. Moreover, in [92] they show an interesting result about the stability of *SALSA* against the *TKC effect* (Tightly Knit Community effect). A *TKC* is a small but highly interconnected set of pages. Roughly, the *TKC effect* occurs when such a community scores high in link-analyzing algorithms, even though the pages in the *TKC* are not authoritative on the topic, of pertain to just one aspect of the topic. They demonstrated that the Mutual Reinforcement approach (used by *HITS*) is vulnerable to this effect, and will sometimes rank the pages of a *TKC* in unjustified high positions.

2.2 Parallel and Distributed Techniques

Nowadays the mostly used parallel architectures are: *Cluster of PC* and the *Grid*. The former consist of a collection of interconnected stand-alone computers working together as a single, integrated computing resource [22, 12]. The latter is a distributed computing infrastructure for advanced science and engineering. The underlining problems of *Grid* concepts consist of coordinating resource sharing and problem solving in dynamic, multi-institutional, virtual organizations [57, 76].

The remainder of this section shows the current research activities in the field of Parallel and Distributed WSEs. In Subsection 2.2.1 we present an analysis of the mostly known techniques used to parallelize the crawling phase. In Subsection 2.2.2 we review the current technologies concerning the design of parallel indexers and, finally, in Subsection 2.2.3 we discuss parallel query analyzers.

2.2.1 Parallel Crawling

As the size of the Web grows, it becomes more difficult to retrieve the whole or a significant portion of the Web using a single process. Therefore many WSEs run multiple

crawler processes in parallel. We refer to this type of crawler as a *parallel crawler*.

Only few works discuss the architecture of parallel crawlers [108, 73, 33, 16, 144].

In [108], Heydon and Najork describe the software architecture of *Atrax* the distributed version of *Mercator* [73]: a scalable, and extensible web crawler. Crawling is performed by multiple worker threads. Each thread, repeatedly performs the steps needed to download and process a document. All *Mercator* threads are run in a single process. However, *Mercator* can be configured as a multi-process distributed systems. In this configuration, one process is designated the *queen*, and the others are *drones*. Both the queen and the drones run worker threads, but only the queen runs a background thread responsible for logging statistics, terminating the crawl, and initiating checkpoints. Moreover, in its distributed configuration the state of a *Mercator* crawl is fully partitioned across the queen and drone processes; there is no replication of data. In a distributed crawl, when a link has been extracted it is checked to see if it is assigned or not to this process. If not it is routed to the appropriate peer process. Since about 80% of links are relative, the vast majority of discovered URLs remain local to the crawling process that discovered them. *Mercator* (*Atrax*) was written in Java, which gives flexibility through pluggable components but also posed a number of performance problems that have been addressed by the authors.

In his PhD thesis, Cho presents various challenges in the development of an effective Crawler [33]. In particular he addresses the parallelization of the Crawling phase. The goal is to propose some guidelines for crawler designers, helping them selecting operational parameters like: number of crawling processes, or inter-process coordination and communication schemes. The author considers a general architecture of a parallel crawler as composed by several crawling processes named *C-procs*. Each *C-proc* performs the basic tasks that a single-process crawler conducts. The *C-procs* may be running either on a LAN (*Intra-site Parallel Crawler*) or on the Internet (*Distributed Crawler*).

In [16] the authors present *Trovatore*, an highly scalable distributed web crawler. The software architecture consists of a number of agents, each one delegated to deal with a specified portion of the web domain under investigation. The main components of the crawler are: the *Store*, that deals with the storage of the crawled pages and the checking of the duplicates; the *Frontier*, that retrieves new pages on the basis of the actually fetched pages; the *Controller* that serves as crash-failure detector.

In [144] the authors propose the use of mobile agents to improve the performance of web search engines. The performance gains translate to improved web coverage and freshness of search results. They use an approach consisting of uploading a software agent to participating servers and using this agent to collect pages and sending them to the search engine site. The authors also give explanations of the security issues related to this approach and show that, due to its simplicity, their proposal does not introduce new security concerns. Moreover, security can be enforced by simple conventional techniques which are computationally inexpensive.

2.2.2 Parallel Indexing

Despite their simple structure, the task of building inverted files for very large text collections such as the Web is very expensive. Therefore, faster indexing algorithms are always desirable and the use of parallel or distributed hardware for generating the index is an obvious solution.

An important feature of the IF index organization is that indices generated following this can be easily partitioned. In particular depending on the moment the partitioning phase is done one can devise two different partitioning strategies [10, 151]. The first approach requires to horizontally partition the whole inverted index with respect to the lexicon, so that each query server stores the inverted lists associated with only a subset of the index terms. This method is also known as *term partitioning* or *global inverted files*. The other approach, known as *document partitioning* or *local inverted files*, requires that each query server becomes responsible for a disjoint subset of the whole document collection (vertical partitioning of the inverted index). Following this last approach the construction of an IF index become a two-staged process. In the first stage each index partition is built locally and independently from a partition of the whole collection. The second phase is instead very simple, and is needed only to collect global statistics computed over the whole IF index.

In [120] the authors address the problem of constructing inverted files using distributed algorithms for building global inverted files for large collections distributed across a high-bandwidth NOW². The three distinct algorithms are : *LL*, *LR*, and *RR*. The *LL* algorithm is based on the idea of computing first the local inverted files at each machine (using a sequential algorithm) and merging them in a subsequent phase. The *LR* and *RR* algorithm are more sophisticated and also superior; they use a mixing of local computation of inverted lists and remote fusion of these for *LR*, and use all remote operations for the *RR* method. The authors argue that the *LR* algorithm is the best in practice because it presents performance comparable to that of the *RR* algorithm but is simpler to implement.

In [102] the authors introduces a novel pipelining technique for structuring the core index-building system that substantially reduces the index constructing time. They demonstrate that for large collections, the pipelining technique can speed up index construction by several hours. They propose and compare different schemes for storing and managing inverted files using an embedded database system. They show that an intelligent scheme for packing inverted lists in the storage structures of the database can provide performance and storage efficiency comparable to tailored inverted file implementations. Finally, they identified the key characteristics of methods for efficiently collecting global statistics from distributed inverted indexes. The *ME* strategy consists of sending local information as the computation executes; the *FL* strategy consists of sending local information during the flush of sorted runs.

²Network Of Workstation.

2.2.3 Efficient Query Brokering

A common software architecture for parallel IRSs follows the *Master/Worker* model. In this model the Workers are the actual search module which receive queries from and return results to the Master that, in this schema, is known as the *Query Broker* (QB).

Since realistic WSEs usually manage distinct indexes, the only way to ensure timely and economic retrieval is designing the QB module so that it forward a given query only to the workers managing documents related to the query topic. The *Collection Selection* technique play a fundamental role in the reduction of the search space. Particular attention should be paid in using this technique since it could results in a loss of relevant documents thus obtaining dramatic effectiveness performance degradations.

In the last ten years a large number of research works dealt with the collection selection problem [97, 23, 24, 39, 40, 47, 51, 62, 60, 61, 63, 64, 68, 67, 66, 69, 72, 118, 77, 82, 87, 90, 95, 106, 115, 117, 127, 147, 153, 154, 156].

The most common approaches to distributed retrieval exploits a number of heterogeneous collections grouped by source and time period. A *collection selection index* (CSI) summarizing each collection as a whole, is used to decide which collections are most likely to contain relevant documents for the query. Document retrieval will actually only take place on such collections. In [72, 40] several selection methods have been compared. The authors showed that the naïve method of using only a collection selection index lacks in effectiveness. Many proposals try to improve both the effectiveness and the efficiency of the previous schema.

In [106] Moffat *et al.*, use a centralized index on blocks of B documents. For example, each block might be obtained by concatenating documents. A query first retrieves block identifiers from the centralized index, then searches the highly ranked blocks to retrieve single documents. This approach works well for small collections, but causes a significant decrease in precision and recall when large collections have to be searched.

In [67, 68, 69, 147] H. Garcia–Molina *et al.* propose GLOSS, a broker for a distributed IR system based on the boolean IR model that uses statistics over the collections to choose the ones which better fits the user's requests.

The authors of GLOSS made the assumption of independence among terms in documents so, for example, if term A occurs f_A times and the term B occurs f_B times in a collection with D documents, than they estimated that $\frac{f_A}{D} \cdot \frac{f_B}{D} \cdot D$ documents contain both A and B .

In [66] the authors of GLOSS generalize their ideas to vector space IR systems (gGLOSS), and propose a new kind of server hGLOSS that collects information for several GLOSS servers and select the best GLOSS server for a given query.

In [24] the authors compare the retrieval effectiveness of searching a set of distributed collections with that of searching a centralized one. The system they used to rank collections is an inference network in which the leaves represent document collections, and the representation nodes represent the terms that occur in the collection. The probabilities that flow along the arcs can be based upon statistics that are analogous to tf and idf in classical document retrieval: document frequency df (the number of documents containing

the term) and inverse collection frequency *icf* (the number of collections containing the term). They call this type of inference network a *collection retrieval inference network*, of *CORI net* for short. They found no significant differences in retrieval performance between distributed and centralized searching when about half of the collections on average were searched for a query. Since the total number of collections was small (*approx 17*), and the percentage of collection searched was high, their results may not reflect the true retrieval performance in a realistic environment.

In [156] (*CVV (Cue-Validity Variance)*) a new collection relevance measure is proposed. This measure is founded on the concept of *cue-validity* of a term in a collection, and measures the degree to which the term distinguishes documents in the collection itself. The results show that the effectiveness ratio decreases as very similar documents are stored within the same collection.

In [154] collection selection strategies using Cluster-based language models have been investigated. Xu *et al.* proposed three new methods of organizing a distributed retrieval system based on the basic ideas presented before. This three methods are *global clustering*, *local clustering*, and *multiple-topic representation*. In the first method, assuming that all documents are made available in one central repository, a clustering of the collection is created; each cluster is a separate collection that contains only one topic. Selecting the right collections for a query is the same as selecting the right topics for the query. This method is appropriate for searching very large corpora, where the collection size can be Terabytes. The next method is *local clustering* and it's very close to the previous one except the assumption of a central repository of documents. This method can provide competitive distributed retrieval without assuming full cooperation among the subsystems. The disadvantage is that its performance is slightly worse than that of global clustering. The last method is *Multiple-topic Representation*. In addition to the constraints in local clustering, the authors assume that subsystems do not want to physically partition their documents into several collections. A possible reason is that a subsystem has already created a single index and wants to avoid the cost of re-indexing. However, each subsystem is willing to cluster its documents and summarize its collection as a number of topic models for effective collection selection. With this method a collection corresponds to several topics. Collection selection is based on how well the best topic in a collection matches a query. The advantage of this approach is that it assumes minimum co-operation from the subsystem. The disadvantage is that it is less effective than both global and local clustering.

In [153] the authors evaluate the retrieval effectiveness of distributed information retrieval systems in realistic environments. They propose two techniques to address the problem. One is to use phrase information in the collection selection index and the other is query expansion. In [47] Dolin *et al.* present Pharos, a distributed architecture for locating diverse sources of information on the Internet. Such architectures must scale well in terms of information gathering with the increasing diversity of data, the dispersal of information among a growing data volume. Pharos is designed to scale well in all of these aspects, to beyond 10^5 sources. The use of a hierarchical metadata structure greatly enhances scalability because it provides for hierarchical network organization.

2.2.4 Peer To Peer

Peer-to-peer (*P2P*) systems are distributed systems in which nodes of equal roles and capabilities exchange information and services directly with each other.

In recent years, P2P emerged as a popular way to share huge volumes of data. The usability of these systems depends on effective techniques to find and retrieve data. For example, the Morpheus [135] multimedia file-sharing system reported over 470,000 users sharing a total of 0.36 petabytes of data as of October 26, 2001. Sharing such large volumes of data is made possible by distributing the main costs – disk space for storing the files and bandwidth for transferring them – across the peers in the network. In addition to the ability to pool together and harness large amounts of resources, the strengths of existing P2P systems (e.g., [136, 134, 149, 135]) include self-organization, load-balancing, adaptation, and fault-tolerance. Because of these desirable qualities, many research projects have been focused on understanding the issues surrounding these systems and improving their performance (e.g., [123, 89, 43]).

The key to the usability of a data-sharing P2P system, and one of the most challenging design aspects, is the design of efficient techniques for search and retrieval. The best search techniques for a given system depends on the needs of the application. For storage or archival systems focusing on availability, search techniques such as Chord [143], Pastry [122], Tapestry [157], and CAN [119] are well-suited, because they guarantee location of content within a bounded number of hops. To achieve these properties, these techniques tightly control the data placement and topology within the network, and currently support only search by object identifier.

In systems where persistence and availability are not guaranteed or necessary, such as Gnutella [134], Freenet [149], Napster [136], and Morpheus [135], search techniques can afford to have looser guarantees. However, because these systems are meant for a wide range of users that come from non-cooperating organizations, the techniques cannot afford to strictly control the data placement and topology of the network. Friends may want to connect to other friends, while strangers may not want to store data (potentially large amounts) on behalf of each other. Also, these systems traditionally offer support for richer queries than just search by identifier, such as keyword search with regular expressions. Search techniques for these “loose” systems must therefore operate under a different set of constraints than techniques developed for persistent storage utilities.

The main problem with the current search techniques is that “loose” P2P systems tend to be very inefficient, either generating too much load on the system, or providing for a very bad user experience.

The original motivation for P2P research was developing file-sharing P2P systems such as Gnutella [134], Napster [136], Freenet [149], and Morpheus [135]. All four of these systems are file-sharing systems that do not guarantee availability.

Napster is not a pure P2P system, but rather a *hybrid* one containing some centralized components; performance of hybrid P2P systems is explored in [155]. Morpheus is a newer, very popular system which has an architecture partway between Napster’s and Gnutella’s. “Super-peer” nodes act as a centralized resource for a small number of clients,

but these super-peers than connected the each other to form a pure P2P network.

Other search techniques for “loose” systems include [2, 41, 137]. The search technique proposed in [2] makes the nodes index the content of other nodes in the system, thereby allowing queries to be processed over a large fraction of all content while actually visiting just a small number of nodes.

Works [41] and [137] propose that each node maintain metadata that can provide “hints” to which nodes contain data relevant for the current query. Query messages are routed by nodes making local decision based on this hints. Hints in [41] are formed by building *summaries* of the content that is reachable via each connection a node has. Hints in [137] are formed by learning user preferences. By observing the behavior of the users – for example, noting which nodes the user choose to download from – a users’s client can learn which nodes the user believes to be an *authority* on certain topics. Future queries on those topics can subsequently be routed to these authority nodes.

Search techniques for systems with strong guarantees on availability include Chord [143], CAN [119], Pastry [122], and Tapestry [157]. These four techniques are quite similar in concept, but differ slightly in algorithmic and implementation details. In Chord, Pastry and Tapestry, nodes have a numerical identifier, while in CAN, nodes are assigned regions in a d -dimensional identifier space. A node is the responsible for owning objects, or pointers to objects, whose identifiers map to the node’s identifier or region. Nodes also form connections based on the properties of their identifiers. With these deliberately formed connections and intelligent routing, a system can locate an object by its identifier within a bounded number of hops. These techniques perform quite well for the systems they were intended for ([43, 123, 89]).

Another interesting project on which some of the Sun’s researchers are currently working on is *JXTA* [84]. *JXTA* is a set of open, generalized P2P protocols that allow any connected device (cell phone, PDA, PC, server) on the network to communicate and collaborate.

Project *JXTA* has a set of objectives that are derived from what we perceive as short-coming of many P2P systems in existence or under development:

- Interoperability;
- Platform Independence; and
- Ubiquity.

JXTA relies heavily on XML [138] to achieve its goals. Messages that are exchanged among the peers are in XML format, which makes *JXTA* platform independent. By separating protocol from specific language bindings, the platform is also technically language independent, although it has initially been implemented using Java. The *JXTA Shell* (written in Java) is a sample application that relies on the *JXTA* platform, but it also shows many features of the platform.

The components of *JXTA* are the results of careful design phases. They provide the minimal requirements for a generic P2P network, stripping it of all the policy-specific

logic and components. This leaves only the building-block constituents that almost all P2P applications can use, regardless of their intended users and specific implementation. One of the key issue for the JXTA core community is to ensure that this generic applicability remains true. The JXTA components enable and facilitate the simple fabrication of P2P applications without imposing unnecessary policies or enforcing specific application operational models.

The JXTA platform contains three layers. These layers in order from lowest to highest are the *core*, the *services*, and the *application*.

The core is the layer in which the peer ID is assigned and protocols are defined and managed. JXTA is made up of a set of six protocols, which allow for the most basic peer discovery and communication. These protocols are language independent. The six protocols are peer *discovery*, peer *information*, peer *membership*, peer *resolver*, peer *binding*, and peer *endpoint*. The protocols provide the basic functions of creating groups, finding groups, joining and leaving groups, monitoring groups, talking to other groups and peers, and sharing information. These protocols are nothing more than XML advertisements and messages among peers. It is in the core that these protocols are encapsulated.

The services layer provides basic functionality to the applications layer. This functionality includes *indexing*, *searching*, and *file sharing*. Applications may or may not decide to use the services layer, but when used, it provides a library-like functionality. An example of a service is Sun's Content Management System (CMS) that provides a generic way for different applications to share files on peers so that searches can be performed against them [88].

The application layer is the place where the p2p applications reside. It is here that applications such as a chat programs, email, auctions, and even storage systems will be found.

Since JXTA is still being developed and refined, the binding is not yet complete. As Sun developers and the community dig into JXTA, they will be finding it necessary to make changes to enhance the framework. With the changes will most definitely come some changes in the interfaces, implementations, and protocols. These changes will result in applications that have already been written needing to be modified in order to account for the changes.

Another problem JXTA faces involves the widespread use of firewalls. Currently, firewalls are a threat to JXTA's usability. A peer inside a firewall is unable to discover peers outside of the firewall. However, there are ways around this problem; for example if you know the address of a peer which is acting as a rendezvous peer, then you can use that peer to locate other peers outside the firewall. JXTA's developers admit that getting around the firewall is a problem and are currently spending much time and effort researching this problem in hopes of coming up with a more ideal solution.

Another interesting relationship is among *P2P* and embedded systems such as *mobile* and *handheld* computers.

This is a new research field and many interesting aspects have to be investigated.

In [70] the authors examine a set of requirements for robust peer computing, propose a small set of architectural principles designed to meet those requirements, and discuss

the systematic exploitation of those principles in the context of *Magi*, a general purpose, peering infrastructure for both embedded and enterprise applications.

The main characteristics of *Magi* is that it uses the HTTP protocol, peers link together in networks to share information and services.

Each *Magi* peer provides a simple core set of services that handles interaction with the network, manages information about the location, status, and access privileges of “*buddies*”³, and supports the dynamic extension of *Magi*. Extension modules may depend on these capabilities being available in any *Magi* peer into which the module is loaded.

³“*Buddy*” stands for peers in the *Magi* environment.

Chapter 3

Caching Search Engine Query Results

Abstract

This chapter discusses the design and implementation of an efficient two-level caching system aimed to exploit the locality present in the stream of queries submitted to a Web search engine. Previous works showed that there is a significant temporal locality in the queries, and demonstrated that caching query results is a viable strategy to increase search engine throughput. We enhance previous proposals in several directions. First we propose the adoption of a new caching strategy. We called this new policy SDC (Static and Dynamic Caching). Basically, it consists of storing the results of the most frequently submitted queries in a *fixed-size read-only* portion of the cache. The remaining portion of the cache is used by the queries that cannot be satisfied by the read-only one according to a single cache replacement policy. Moreover, we show that search engine query logs also exhibit spatial locality, since users often require subsequent pages of results for the same query. SDC also take advantage of this type of locality by exploiting a sort of *speculative* prefetching strategy. We experimentally demonstrate the superiority of our SDC over other policies. We measured the hit-ratio achieved on three large query logs by varying the size of the cache, the percentage of read-only entries, and the replacement policy used for managing the dynamic cache. Finally, we propose an implementation optimized for concurrent accesses, and we accurately evaluate its scalability.

Caching is a very effective technique to make scalable a service that distributes data to a multitude of clients. As suggested by many researchers, caching can also be used to improve the efficiency of a Web Search Engine (WSE) [152, 99, 125, 94]. This is motivated by the high locality present in the stream of queries processed by a WSE, and by the relatively infrequent updates of WSE indexes that allow us to think of them as mostly read-only data.

3.1 Introduction

Nowadays, WSEs are commonly used to find information and navigate in the Web. In the last years we have observed an impressive growth in the number of pages indexed as well as in the number of queries submitted to the most popular WSEs, thus requiring WSEs to be designed in a scalable way.

WSE caching, similarly to Web page caching, can occur at several places, e.g. on the client side, on a proxy, or on the server side. Caching on either the client or the proxy has the advantage of saving network bandwidth. Caching on the server side, on the other hand, has the advantage of improving the shareness of query results among different users. Moreover, caching at this level has the effect of saving I/O and computational resources used by the WSE to compute the page of relevant results to be returned to a user. In fact, consider that, in order to prepare a page of results, we have to intersect inverted lists that can be distributed, and to globally rank the results to decide which are the most relevant ones. On the other hand, cache results are cheaper to retrieve since it, usually, involves just a look-up operation on a search data structure.

One of the issues related to a server-side cache is the limited resources usually available on the WSE server, in particular the RAM memory used to store the cache entries. However, the architecture of a scalable, large-scale WSE is very complex and includes several machines which take care of the various sub-tasks involved in the processing of user queries [112, 20]. Figure 3.1 shows the typical architecture of a modern large-scale WSE placed behind an http server. We can see a distributed architecture composed by a farm of identical machines running multiple WSE CORE modules, each of which is responsible for searching the index relative to one specific sub-collection of documents. This organization of the index is called *Local Inverted File* or *Document Partitioning*, in contrast to a *Global Inverted File* or *Term Partitioning* in which a complete index is horizontally split so that different index partitions refer to a subset of the set of distinct terms of the collection. In front of these searcher machines we have an additional machine hosting the MEDIATOR/BROKER. This module has the task of scheduling the queries to the various searchers, and of collecting the results returned back. The mediator then orders these results on the basis of their relevance, and produces a ranked vector of document identifiers (DocIDs), e.g. a vector composed by 10 DocIDs. These DocIDs are then used to get from the URL/SNIPPETS SERVER the associated URLs and page snippets to include in the html page returned to the user through the http server. Note that multi-threading is exploited extensively by all these modules in order to process concurrently distinct queries. Within this architecture the RAM memory is a very precious resource for the machines that host the WSE CORE, which perform well only if the mostly accessed sections of their huge indexes can be buffered into the main memory. Conversely, the RAM memory is a less critical resource for the machine that hosts the mediator. This machine can thus be considered as an ideal candidate to host a server-side cache. The performance improvement which may derive from the exploitation of query results caching at this level is remarkable. Queries resulting in cache-hits are in fact promptly served thus enhancing WSE throughput, but also the queries whose results are not found in cache ben-

efit substantially due to the lower load on the WSE and the consequent lower contention for the I/O, network and computational resources.

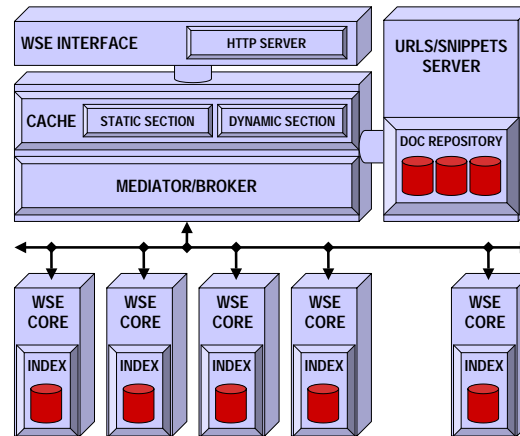


Figure 3.1: Architecture of a large-scale, distributed WSE hosting a query results cache.

In this chapter, we are interested in studying the design and implementation of such a server-side cache of query results. In particular, we will analyze the performance of a one-level cache in terms of hit-ratio and throughput.

Starting from an accurate analysis of the content of three real query logs, we propose a novel replacement policy (called SDC- Static and Dynamic Cache) to adopt in the design of a fully associative cache of query results. According to SDC, the results of the most frequently accessed queries are maintained in a fixed size set of statically locked cache entries. This set is called *Static Set* and it is rebuilt at fixed time intervals using the statistical data coming from the WSE usage data. When a query arrives at SDC if it cannot be satisfied by the Static Set it competes for the use of a *Dynamic Set* of cache entries.

We experimentally demonstrated the superiority of SDC over other caching policies proposed elsewhere, by evaluating both the hit-ratio and the throughput achieved on the three query logs by varying the size of the cache, the percentage of cache entries of the Static Set, and the replacement policy used for managing the Dynamic Set. Moreover, we showed that WSE query logs exhibit not only temporal locality, but also a limited spatial locality, due to requests for subsequent pages of results. Furthermore, our caching system exploits a sort of *Speculative Prefetching Strategy* that, differently from other prefetching proposals (e.g. [94]), tries to maintain a low overhead on the underlying WSE Core. In fact, while server-side caching surely reduces the load over the core query service of a WSE and improves its throughput, prefetching aims to increase the cache hit-ratio and thus the responsiveness (from the point of view of each single user) of the WSE, but may involve a large overhead on the same core query service. So an accurate study of trade-offs of prefetching is necessary, and we addressed this in the experimental section of the

chapter by analyzing pros and cons of different prefetching strategies.

In the last part of the chapter, we will also discuss actual implementation issues and we will present a concurrent implementation of a cache system using SDC. To show the benefits of our caching system, we will accurately assess the scalability of the whole system with respect to the number of concurrent threads accessing the cache data structures.

The rest of this chapter is organized as follows. In Section 3.2 a review of the research carried out during these years on caching WSE results is shown. Section 3.3 describes the query logs used and discusses the results of a statistical analysis conducted on their contents. In Section 3.4 we describe our novel policy, whereas Section 3.5 shows the results of the simulations performed on the different query logs. Section 3.6 describes an efficient implementation of the cache and, finally, Section 3.7 present the conclusions of this chapter.

3.2 Related Work

Several complex cache replacement policies have been proposed so far. These policies try to find a trade-off between two criteria: the recency of references and their global frequency. In many cases these policies have been used in Web Server, Disk, or CPU caches.

Besides these, a quite large number of works have addressed the problem of studying the behavior of WSEs users [100, 94, 55, 124, 152, 142, 80, 140, 74, 131, 96, 56, 141, 30, 44, 79, 78, 93].

In one of the first papers discussing the behavior of WSEs users [74], Hoelsher analyzes the query log of a popular German search engine: Fireball¹. The log contained about 16 millions of queries submitted on July 1996. The main focus of the paper is showing the characteristics of the queries submitted by the users. The main interesting result shows that a large part of the queries (about 59%) look just at the first page of results, while the 79% of the users look at no more than three pages of results².

Henzinger *et al.* in [131] analyze a very large query log of the Altavista search engine. The query log analyzed contains about a billion queries submitted in a period of 42 days. The exhaustive analysis presented by the authors pointed out several interesting results. Tests conducted include the analysis of the query sessions for each user, and the correlation of search terms among the submitted queries. As in the previous work, the results show that the majority of the users (in this case about 85%) display only the first page of results. A new, and quite interesting, result shows that the 77% of the users' session end-up just after the first query has been submitted.

In his work, Markatos analyzes a million queries contained in a log of the Excite WSE [99]. The paper describes the characteristics of the query log and evaluates both the temporal, and the spatial locality. The experiments show the existence of locality in the submission of the queries to the WSE. In particular, nearly a third of the submitted

¹<http://www.fireball.de>

²A page of results, usually, contains ten distinct results.

queries are successively resubmitted by either the same, or other users. Furthermore, in this paper Markatos shows that the plot of the 1,000 popular queries in the log conforms to a zipfian distribution. This last results is confirmed also by Saraiva *et al.* who plotted the frequency of submission of a collection of 100,000 queries submitted to TodoBR, a Brazilian search engine [125].

One of the last published work which is focused on determining the attitude of WSEs users is [94]. In this paper the authors present an analysis of a query log containing around 7.2 millions queries submitted to the Altavista search engine in a week of 2002. In the trace considered, they discovered that about the 63.5% of the views were for the first page of results. On the other hand, the views accounted for the second page of results were measured to be around the 11.7% of the total. Another statistical fact discovered is that the topic popularity, i.e. the number of time a particular query is requested, follows a *Zipf's Law* distribution. Practically speaking, a large number of queries (over the 67%) are requested only once, while very few of them are requested multiple times. In our work, we show the same trend by examining other three real query logs: *Excite*, *Tiscali*, and *Altavista*. The analysis we made on our logs, in fact, evidenced the same Zipfian's behavior for the distribution of query frequencies.

Beside that quite a large number of studies exists on evaluating the query logs characteristics, just a few of them also propose effective techniques to exploit the locality present in the stream of WSEs requests[99, 125, 94, 55].

In the paper by Markatos [99] he presented, for the first time, a caching system aimed to reduce the response time of a WSE. He implemented several existing caching policies and compared the hit-ratio obtained for each one of them. He tested his implementation on query log composed by queries submitted to Excite. He found that the hit-ratio values for the considered log, were quite good. Nonetheless, he did not present any policy tailored on the statistical properties found on the Excite log. Moreover, Markatos did not consider the possibility of introducing a prefetching strategy in order to prepare the cache to answer the requests for the following results pages.

The second work published describing a real caching system for WSEs is by Saraiva *et al.* [125]. In their work, the authors, show a two-level caching system which try to enhance the responsiveness of a hierarchically-structured search engine (i.e. like the one we consider in the present document, see Figure 3.1). The first-level cache consists of a classical caching system implementing an *LRU* policy. The second-level cache, instead, is intended to store each single posting list of each term contained into the query string. For example, for the query "caching system" the second-level cache will separately store the posting lists' entries for both the terms "caching", and "systems". The main interesting item in this work is that the authors experimented their cache by measuring the overall throughput when either a two-level cache was, or was not adopted. Even though at a low request-rate the second-level cache did not produce any increment in the throughput, when the request-rate increase the second-level can effectively help the disk in serving the requests thus increasing the overall throughput. Also Saraiva *et al.* did not consider the use of any prefetching strategy in their work.

At the WWW12 conference, Ronny Lempel and Shlomo Moran published a paper

where *PDC* (Probabilistic Driven Caching), a new caching policy, was presented. The idea behind *PDC* is to associate a probability distribution to all the possible query that can be submitted to a WSE. The distribution is built over the statistics computed on the previously submitted queries. For all the queries that have not previously seen, the distribution function evaluates to zero. This probability distribution, is used to compute a priority value that is exploited to maintain an importance ordering among the entries of the cache. In practice, the higher the probability of a query to be submitted the higher it is ranked within the cache once it is actually appeared. Indeed, the probability distribution is used only in the case of queries requesting the pages subsequent the first. For the first page of results a simple *SLRU* policy is used. Furthermore, *PDC* adopt prefetching to anticipate users' requests. Note that *PDC*, also consider a model of users' behavior. Within this model, a query-session starts with a request to the WSE. At this point two ways are possible: s/he can either submit a *follow-up* query (i.e. a query requesting the successive page of results), or s/he can give up and possibly start a new session by submitting a different query. A session is considered over, if no follow-up query appears within τ seconds. This model is respected in *PDC* by demoting the priorities of the entries of the cache referring to the queries submitted more than τ seconds ago. Note that the caching policy for the queries requesting the second, and over, page of results are ordered following a priority computed using the statistical data available. They evict from the section the entry which has the lower priority only if the ready-to-enter query has a priority greater than that. Conversely, the queries referring to the first page of results are managed by a separate *SLRU* cache. The results on a query log of Altavista containing queries submitted during a week of 2001, are very good. Using a cache of 256,000 elements using *PDC* and prefetching 10 pages of results, the authors obtained a hit-ratio of about 53,5%. Unfortunately the policy seems to be quite expensive in terms of time-to-serve for each request (in particular those causing a cache miss).

In this work we will try to overcome all the issues pointed out in this section. We will propose, in fact, SDC a novel caching policy which at the same cost of the *LRU* (or *SLRU*) policy (or even lower) will obtain hit-ratios that, in many cases, are better than those of *PDC*. In particular, we will see that, in principle, SDC can be combined with any other existing policies³. In fact, with all the policies we considered in our experiments, SDC has always brought to performance enhancements. Furthermore, differently from other works which used only one log coming from a single WSE, we validated our results on three different query logs, coming from three different WSE, and referring to three different periods of time: a single day, a week, and a month.

3.3 Analysis of the query logs

In order to evaluate the behavior of different caching strategies we used query logs from the Tiscali, EXCITE, and Altavista search engines. In particular we used *Tiscali*, a trace

³Implementing SDC using a logarithmic policy will make the SDC behavior logarithmic too.

Table 3.1: Main characteristics of the query logs used.

Query log	queries	distinct queries	date
<i>Excite</i>	2,475,684	1,598,908	September 26 th 1997
<i>Tiscali</i>	3,278,211	1,538,934	April 2002
<i>Altavista</i>	6,175,648	2,657,410	a week of 2001

of the queries submitted to the Tiscali WSE engine (www.janas.it) on April 2002, *Excite*, a publicly available trace of the queries submitted to the EXCITE WSE (www.excite.com) on September 26th 1997, and *Altavista* a query log containing queries submitted to Altavista on a week of 2002⁴. Each record of a query log refers to a single query submitted to the WSE for requesting a *page* of results, where each page contains a fixed amount of URLs ordered according to a given rank. All query logs have been preliminarily cleaned by removing the useless fields. At the end of this preprocessing phase, each entry of a query log has the form (*keywords*, *page_no*), where *keywords* corresponds to the list of words searched for, and *page_no* determines which page of results is requested. We further normalize the query log entries by removing those referring to requests of more than 10 results-per-page. In particular, since a WSE globally reorders the results according to a given rank, the top 10 results will be included in page 1, the next 10 results in page 2, and so on.

Table 3.1 reports the main characteristics of the query logs used. While about the 46% of the total number of queries appearing in the relatively recent Tiscali and Altavista logs are distinct, in the Excite log this percentage increases up to 64%. Therefore, only looking at the numbers of distinct queries appearing in the three logs, we could deduce that the locality found in the Excite log, i.e. the oldest one, might be less than in the Tiscali ones, since only the 36% (about 54% in the Tiscali and Altavista logs) of all its queries corresponds to re-submissions of previously submitted queries.

3.3.1 Temporal locality

The plots reported in Figure 3.2 assess the temporal locality present in the query logs using a log-log scale. In particular Figure 3.2.(a) plots the number of occurrences within each log of the most popular queries, whose identifiers have been assigned in decreasing order of frequency. Note that, in all the three logs, more than 10,000 different queries are repeated more than 10 times. Since the number of occurrences of a given query is a measure that might depend on the total number of records contained in the logs, to better highlight temporal locality present in the logs we also analyzed the time interval between successive submissions of the same query. The rationale is that if a query is repeatedly submitted within a small time interval, we can expect to be able to retrieve its results even from a cache of small size. Note that according to the experiments performed in previous works, the trend of the three curves follows that of a *Zipf's law*. We briefly recall that the

⁴We are very grateful to Altavista for kindly agreeing to make this query log publicly available.

occurrences y of a phenomenon is said to follow a *Zipf's law* if $y = Kx^{-\alpha}$, where x is the rank of the phenomenon itself. In our case K varies according to the query log analyzed, while $\alpha \approx 0.66$ in all the three cases.

Figure 3.2.(b) reports the results of this analysis. For each query log we plotted the cumulative number of resubmissions of the various queries as a function of the time interval (expressed as a distance measured in number of queries). Once more the results are encouraging: in the *Tiscali* log for more than 350,000 times, the time interval between successive submissions of the same query is less than 100; in the *Altavista* log this temporal locality is slightly smaller than that in *Tiscali* but, again, for more than 150,000 times the time interval is still less than 100. In the case of the *Excite* log we encountered a lower temporal locality. However, also in this log for more than 10,000 times a given query is repeated at a distance lower than 100.

We think that the lower locality present in the *Excite* log is mainly due to its oldest age. It contains several long queries expressed in natural language like "*Where can I find information on Michael Jordan*". In the query above the first six terms are meaningless, while the only informative keywords are the last two. Such kind of queries can be considered a symptom of the poor capacity of the users to interact with a WSE six years ago. Although we tried to clean the *Excite* queries by removing *stopwords*, i.e. by eliminating meaningless terms like articles, adverbs and conjunctions, the query above still resulted syntactically different from the simpler query "*Michael Jordan*" even if the same results may be considered relevant for both the queries.

3.3.2 Spatial locality

As discussed above, several works analyze WSE query logs in order to determine behaviors and preferences of users [74, 131, 142, 79, 141, 94]. Although different in some assumptions and in several conclusions, these works highlight that WSE users in most cases submit short queries and visit only a few pages of results. Estimations reported in these works differ, in some cases, remarkably: depending on the query log analyzed, percentages ranging from 28% to 85% of user searches only require the first page of results, so that we can expect that from 15% up to 72% of user searches retrieve two or more pages of results. This behavior involves the presence of significant spatial locality in the stream of queries processed by a WSE. Given a query requesting a page of relevant results with respect to a list of keywords, with high probability the WSE will receive a request for one of the following pages within a small time interval. To validate this consideration we measured the spatial locality present in our query logs. Figure 3.3 reports the results of this analysis. In particular, Figure 3.3.(a) plots the percentage of queries in each log file as a function of the index of the requested page. As expected, most queries require the first page of results. On the other hand, while there is a huge difference between the number of queries requesting the first and the second page, this difference becomes less significant when we consider the second and the third page, the third and the fourth, and so on. This may reflect different usages of the WSE. When one submits a focused search, the relevant result is usually found in the first page. Otherwise, when a generic query is

submitted, it is necessary to browse several pages in order to find the relevant information. To highlight this behavior, Figure 3.3.(b) plots the probability of the occurrence of a request for the i -th page, given a previous request for the $(i - 1)$ -th one. As it can be seen, this probability is low for $i = 2$, while it increases remarkably for higher values of i .

3.3.3 Theoretical upper bounds on the cache hit-ratio

From the analysis of the three query logs we can devise some theoretical upper bound on the maximum performance, in terms of hit-ratio, attainable. To do so we must relax some constraints on the problem. In particular, we will suppose that no eviction will occur from the cache thus supposing the availability of an infinite sized cache.

Let us starting by considering the case where prefetching is not used. In this case the minimum miss-ratio attainable by an infinite sized cache, m , is given by considering the number of distinct queries, D , over the total number of queries, Q .

$$m = \frac{D}{Q} \quad (3.1)$$

As a consequence the maximum hit-ratio, H , is given by

$$H = 1 - m = 1 - \frac{D}{Q} \quad (3.2)$$

If we apply Equation 3.2 to the data of Table 3.1 we obtain the following upper bounds:

- *Excite*: 0,3541 (35.41%)
- *Tiscali*: 0,5305 (53.05%)
- *Altavista*: 0,5697 (56.97%)

Things become a little more complicated if we use prefetching. In this case, in fact, the responses to the users' queries are anticipated by prefetching thus it is not possible to analytically compute the hit-ratio bounds. To do so, we must perform another kind of analysis on the query logs. Like in the paper by Lempel and Moran [94], we compute the minimum number of misses as follows. We associate to each possible topic a bitmap of P_{max} bits, where P_{max} is the largest number of pages a query can refer to. Whenever a page i is requested for topic T we set to 1 the correspondent i -th bit in the bitmap of T . At the end of this process, the minimum number of misses in case of prefetching p can be obtained by first subdividing each bitmap into bins of size p and counting the number of bins containing at least a 1. At this point is sufficient to divide this value by the total number of queries. Table 3.2 shows the results computed on the three query logs; in our case we set P_{max} to 32.

As it can be seen in the table above, for large values of prefetching factors we do not experience a correspondent large variations in hit-ratio bounds. In particular, for prefetching factor greater than 4 the variations are in the second decimal digit.

Prefetching factor	Query log			Prefetching factor	Query log		
	<i>Excite</i>	<i>Tiscali</i>	<i>Altavista</i>		<i>Excite</i>	<i>Tiscali</i>	<i>Altavista</i>
2	45.4283	56.652	60.8912	11	47.5763	57.0003	62.7765
3	47.2301	56.8608	61.7167	12	47.5768	57.0006	62.8283
4	47.4871	56.9751	62.078	13	47.5769	57.0007	62.8758
5	47.5442	56.9881	62.2799	14	47.5769	57.0008	62.9182
6	47.5634	56.9936	62.4008	15	47.5771	57.0009	62.9588
7	47.5685	56.9963	62.4886	16	47.578	57.0009	62.9968
8	47.5695	56.9979	62.5664	17	47.578	57.001	63.0329
9	47.5701	56.999	62.6374	18	47.5782	57.001	63.0725
10	47.5705	56.9997	62.716	19	47.5783	57.001	63.1124
				20	47.5783	57.001	63.1657

Table 3.2: Hit-ratio upper bounds for each query log as a function of the prefetching factor.

3.4 The SDC policy

In this section we describe SDC (Static and Dynamic Cache) a novel caching policy. Actually, this is a work extending a previously presented research [55]. The idea that drove the entire design process is the following: *Is it possible to find a policy suitable for caching data which appear in accordance with a Zipf's law distribution, and having a time complexity equal to that of LRU/SLRU?*

SDC is a two-level policy which makes use of two different sets of cache entries. The first level contains the so called *Static Set*. It consists of a set of statically locked entries filled with the most frequent queries appeared in past users' sessions. The Static Set is periodically refreshed. The second level contains the *Dynamic Set*. Basically, it is a set of entries managed by a classical replacement policy (i.e. *LRU*, *SLRU*, etc.).

The behavior of SDC in the presence of a query q is, thus, very simple. First it looks for q in the Static Set, if q is present it returns the associated page of results back to the user. If q is not contained within the Static Set then it proceeds by looking for q in the Dynamic Set. If q is not present, then SDC asks the WSE for the page of results and replaces a page according to the replacement policy adopted.

Note that the idea of using a statically locked cache is present also in the work from Markatos where he studied a pure static caching policy for WSE results and compared it with a pure dynamic ones [99].

The rationale of adopting a static policy, where the entries to include in the cache are statically decided, relies on the observation that the most popular queries submitted to WSEs do not change very frequently. On the other hand, several queries are popular only within relatively short time intervals, or may become suddenly popular due to, for example, un-forecasted events (e.g. the 11th September 2001 attack). Basically, if the queries are distributed following a sort of *Zipf's law* behavior, then we may statically identify a set of queries to insert in the first level of the cache.

The advantages deriving from this novel caching strategy are two-fold. First, SDC

present many interesting capabilities achieving the main benefits of both static and dynamic caching. In fact:

- the results of the most popular of the queries can always be retrieved from the Static Set even if some of these queries might be not requested for relatively long time intervals;
- the Dynamic Set of the cache can adequately cover sudden interests of users.

Second, SDC may enhance performance. In fact, since accesses in the read-only section can be made concurrently without synchronization, this would eventually bring to good performance in a multi-threading environment.

3.4.1 Implementation Issues

First level - Static Set

The implementation of the first level of our caching system is very simple. It basically consists of a lookup data structure that allows to efficiently access a set of $f_{static} \cdot N$ entries, where N is the total number of entries of the whole cache, and f_{static} the factor of locked entries over the total. f_{static} is a parameter of our cache implementation whose admissible values ranges between 0 (a fully dynamic cache) and 1 (a fully static cache). The static cache has to be initialized off-line, i.e., with the results of most frequent queries computed on the basis of a previously collected query log.

Each time a query is received, SDC first tries to retrieve the corresponding results from the Static Set. On a cache hit, the requested page of results is promptly returned. On a cache miss, we also look for the query results in the Dynamic Set.

Second level - Dynamic Set

The Dynamic Set relies on a replacement policy for choosing which pages of query results should be evicted from the cache as a consequence of a cache miss and the cache is full. Literature on caching proposes several replacement policies which, in order to maximize the hit-ratio, try to take the largest advantage from information about recency and frequency of references. SDC surely simplifies the choice of the replacement policy to adopt. The presence of a static read-only cache, which permanently stores the most frequently referred pages, makes in fact recency the most important parameter to consider. As a consequence, some sophisticated (and often computationally expensive) policies specifically designed to exploit at the best both frequency and recency of references are probably not useful in our case. However, since we want to demonstrate the advantage of the SDC policy over the others, we implemented some of these sophisticated replacement policies.

Currently, our caching system supports the following replacement policies: *LRU*, *LRU/2* [111] which applies a *LRU* policy to the penultimate reference, *FBR* [121], *SLRU*

[99], *2Q* [83], and *PDC* [94] which consider both the recency and frequency of the accesses to cache blocks.

The choice of the replacement policy to be used is performed at start-up time, and clearly affects only the management of the $(1 - f_{static}) \cdot N$ dynamic entries of our caching system.

Hereinafter we will use the following notation to indicate the different flavor of SDC. We will use SDC- r_s to indicate SDC with replacement policy r , $f_{static} = s$. For example: SDC-*LRU*_{0.4} means we are referring to SDC using *LRU* as the replacement policy, and $f_{static} = 0.4$. Another example could be the following: SDC-*[LRU/SLRU]*_{0.4} which indicate SDC with a replacement policy chosen among *LRU*, and *SLRU*. Moreover, we will use the jolly character *, to indicate all the possible choice for the parameter replaced by *. So, SDC- $*_s$ will indicate SDC with any replacement policy and $f_{static} = s$; while, SDC- p_* will indicate SDC with the replacement policy p and any value of f_{static} .

3.5 Experiments with SDC

3.5.1 Experimental setup

All the experiments were conducted on a Linux PC equipped with a 2GHz Pentium Xeon processor and 1GB of RAM.

Since SDC requires the blocks of the static section of the cache to be preventively filled, we partitioned each query log into two parts: a *training set* which contains 2/3 of the queries of the log, and a *test set* containing the remaining queries used in the experiments. The N most frequent queries of the training set were then used to fill the cache blocks: the first $f_{static} \cdot N$ most frequent queries (and corresponding results) were used to fill the static portion of the cache, while the following $(1 - f_{static}) \cdot N$ queries to fill the dynamic one. Note that, according to the scheme above, before starting the tests not only the static blocks but also the dynamic ones are filled, and this holds even when a pure dynamic cache ($f_{static} = 0$) is adopted. In this way we always starts from the same initial state to test and compare the different configurations of SDC, obtained by varying the factor f_{static} . (i.e. warm cache, using the terminology in [94]). The most important fact to notice, is that differently from previous works, which validated their results only on a particular query log, we used three different query logs to evaluate SDC performance. Moreover, the queries have been grabbed in different periods and, more important, the duration of these periods is different. While the *Excite* query log refers to a single day, the *Altavista* refers to a week, and the *Tiscali* one which, instead, refers to an entire month. Beside this fact, however, we will see that the behavior of SDC will remain the same for all the three logs.

3.5.2 SDC without prefetching

Figures 3.4 and 3.5 reports the cache hit-ratios obtained on the query logs *Tiscali*, *Excite*, and *Altavista* by varying the ratio (f_{static}) between static and dynamic sets. Each curve corresponds to a different replacement policy used for the dynamic portion of the cache. In particular, f_{static} was varied between 0 (a fully dynamic cache) and 1 (a fully static cache), while the replacement policies exploited were *LRU*, *FBR* [121], *SLRU* [99], *2Q* [83], and *PDC* [94]. The total size of the cache was fixed to 50,000 blocks for a test on a small cache and to 256,000 blocks for the other test.

Several considerations can be done looking at these plots. First, we can note that the hit-ratios achieved are in some cases impressive, although the curves corresponding to different query logs have different peak values and shapes, thus indicating different amounts and kinds of locality in the query logs analyzed. At a first glance, these differences surprised us. After a deeper analysis we realized that similar differences can also be found by comparing other query logs already studied in the literature [74, 131, 142, 79, 141], thus indicating that users' behaviors may vary remarkably from time to time.

Another important consideration is that in all the tests performed SDC remarkably outperformed the other policies, whose performance are exactly those corresponding to a value of $f_{static} = 0$. The best choice of the value for f_{static} depends from the query log considered and, from the period of time on which the training set is based. As we will see, in fact, if we train SDC on a small portion of the query log, we would obtain the best performance in correspondence of smaller f_{static} values. However, for relatively large caches, the hit-ratio values do not varies sensibly with variations of this parameter around the optimum. For example, in the case of a SDC-*LRU*_{0.8} cache of 256,000 entries on the *Altavista* query log has the optimum hit-ratio value of 35.76%. In this case, varying f_{static} from 0.5 to 0.9, the hit-ratios vary between 35.33 and 35.74.

Moreover, as it can be expected, due to the Zipf's law regulating the distribution of query occurrences, the different replacement policies do not impact heavily on the overall hit-ratio. In fact the few queries that are the most frequently submitted are satisfied by the Static Set, while the majority of the queries appearing only once do not benefit from the cache at all. Thus, the Dynamic Set is exploited only by those queries which we can define *Burst Queries* that is those appearing relatively frequently but just for a brief period of time. For those queries, the variations of the hit-ratio figures do not justify the adoption of a policy rather than another.

Different arguments should be done in the case of small caches. For what regards the replacement policies, with the sole exception of *Altavista* where all the policies have comparable performance, the tests demonstrated that SDC-[*LRU/2Q*]* outperforms the others. This behavior is motivated by the presence of the static portion of our cache, which already stores the results of the most frequently referred queries in the past. This seems to penalize replacement policies that consider either both recency and frequency of the accesses to a page, or just the frequency property. In particular, this trend, is also confirmed by the curves of SDC-*FBR** which is a frequency-based policy. SDC-*FBR** is always behind the other curves because the recency is never considered. On the other

hand, *LRU* considers only the recency of the accesses, while the Static Set considers the most frequently accessed queries. As a consequence, a simple (and computational inexpensive) replacement policy like *LRU*, which only considers recency, can be profitably adopted within SDC.

Another quite interesting result is that of *PDC*. In fact, as it can be seen by the plots, whenever *PDC* is used without adopting prefetching, its performance drops considerably: *SDC-SLRU** sensibly outperforms *PDC* (an hit-ratio of 31.19% for a pure *PDC* against 35.76% of *SDC-SLRU*_{0.9}, in the case of the *Altavista* query log and a 256,000 entries cache). On the other hand, in the case of *SDC-PDC**, the hit-ratio figure raises up to 35,69 thus flattening the difference between the various SDC flavors.

Furthermore, *SDC-LRU** has a response time which appears to be much lower than logarithmic policies like, for example, *PDC* or *2Q*. For instance, if we adopt a cache of 256,000 elements. Using *SDC-LRU** will cost 10.05 μ sec per hit and 42.89 μ sec per miss, while using *PDC* will cost 46.18 μ sec per hit and 112.85 μ sec, i.e. about three or four times more than *SDC-[LRU/SLRU]**. Anyway, the most important fact remains the complexity figures of $O(1)$ of SDC vs. $O(\log(n))$ of the others.

To measure the sensitivity of SDC with respect to the size of the cache, Figure 3.6 plots the hit-ratios achieved on the *Tiscali*, *Altavista* and *Excite* query logs as a function of the number of blocks of the cache and the f_{static} parameter. As expected, when the size of the cache is increased, hit-ratios increase correspondingly. In the case of the *Tiscali* log, the hit-ratio achieved is about 37% with a cache of 10,000 blocks, and about 45% when the size of the cache is 50,000. Note that actual memory requirements are however limited: a cache storing the results as an *Html* page and composed of 50,000 blocks requires about 200MB of RAM, while a cache storing just the *DocID* (i.e. a *DocID cache*) and composed by the same number of entries requires less than 5MB!

Once more, these plots show that variations of the f_{static} around the optimum have small impact on the hit-ratio values for large caches.

3.5.3 SDC and prefetching

The spatial locality present in the stream of queries submitted to the WSE can be exploited by anticipating the request for the following pages of results. In other words, when our caching system processes a query of the form (*keywords*, *page_no*), and the corresponding page of results is not found in the cache, it might forward to the core query service WSE an expanded query requesting k consecutive pages starting from page *page_no*, where $k \geq 1$ is the *prefetching factor*. To this end, the queries that arrive at the WSE cache system are thus expanded, and are passed to the core query service of the WSE as (*keywords*, *page_no*, k). Note that, according to this notation, $k = 1$ means that prefetching is not activated.

When the results of the expanded queries are returned to the caching system, the retrieved pages are stored into k distinct blocks of the cache, while only the first page is returned to the requesting user. In this way, if a query for a following page is received within a small time interval, its results can surely be found into the cache.

Figure 3.7 shows the results of the simulation over our query logs in case SDC and prefetching are applied.

As expected, whenever some kind of prefetching is adopted the hit-ratios of the various caching policies considerably increases. For instance SDC- LRU_* on the *Altavista* query log and a cache of 256,000 results in a hit-ratio of 35.76 without prefetching and of 54.20 if prefetching with $k = 10$ is used. That means about 50% of hit-ratio gain.

On the other hand, the prefetching factor cannot indefinitely grow. In fact, let us consider a case of a SDC- LRU_* cache composed by 32,000 elements on the Excite query log. We measured an increase in the hit-ratio values if we keep the prefetching factor under 3, conversely, using larger factors, the hit-ratios start to slowly decrease. Figure 3.8

Needless to say, also the response time of the cache increase as the prefetching factor increases. In fact, for each miss the response time of the cache module is directly proportional to the prefetching factor.

Another big drawback of using large prefetching factors is that on the WSE side the load increase. The cost of resolving a query, in fact, in some cases⁵ is logarithmic with respect to the number of results retrieved [151]. So, an aggressive prefetching strategy might negatively affect the replacement policy adopted and the WSE throughput. By profiling the execution of the tests, we measured that for SDC- $LRU_{0.9}$ cache of size 256,000, in the case of a constant prefetching factor 5, only 11.71% of the prefetched pages are actually referred by the following queries.

In order to maximize the benefits of prefetching, and, at the same time, reduce the additional load on the WSE, we can take advantage from the characterization of the spatial locality present in the logs discussed in Section 3.3. Figure 3.3.(b) shows that, given a request for the i -th page of results, the probability of having a request for page $(i + 1)$ in the future is about 0.1 for $i = 1$, but becomes approximately 0.5 or greater for $i > 1$. Therefore, an effective heuristic to adopt is to prefetch additional pages only when the cache miss has been caused by a request for a page different from the first one. In this way, since the prefetched pages will be actually accessed with sufficiently high probability, we avoid to fill the cache with pages that are accessed only rarely and, at the same time, we reduce the additional load on the core query service of the WSE. Our caching system thus adopts this simple heuristic: the prefetching factor k ($k > 1$) for the expanded query (*keywords*, *page_no*, k) is chosen as follows:

$$k = \begin{cases} K_{MAX} & \text{if } page_no > 1 \\ 2 & \text{otherwise} \end{cases} \quad (3.3)$$

In the formula above, K_{MAX} represent the maximum number of page prefetched by the cache. In practice the heuristic is very simple: whenever the first page is requested expand the query by requesting the first and the second, as well. When the second page is requested return it to the user and ask the underlying WSE for the successive $K_{MAX} - 2$ pages of results.

⁵Often the cost of retrieving the posting lists from the disk dominates the other figures.

Whenever this *speculative* prefetching heuristic is adopted (with $K_{max} = 5$), the percentage of prefetched pages actually referred (i.e. the pages in the cache that are referred by other queries) increases up to 30.3%. We can thus conclude that the proposed *speculative* heuristic constitutes a good trade-off between the maximization of the benefits of prefetching, and the reduction of the additional load on the WSE. Note that, this policy does not result in a hit-ratio drop down. The drawback of this approach, on the other hand, is that it can be effectively used only in a multi-thread context where multiple cache-clients are adopted to serve multiple requests. Otherwise, in the case of a single thread cache, when the second page of result is requested even if the cache contains it, it will be occupied in requesting the following pages thus it could not receive any other request. This, however, is not really a problem since a real world WSE's cache would be, surely, implemented as a multi-threaded process.

3.5.4 Freshness of the Static Set

Caching policies which are driven by statistical data, like SDC and *PDC*, may suffer of problems concerning the freshness of the data from which statistics have been drawn. For this reason we are going to analyze the effect of considering a training set smaller than the one considered previously. As said in Section 3.5.1, in fact, we trained our static cache on the two-third of the query logs. In their paper Lempel and Moran trained the *PDC* statistics on the first million queries. We tested also SDC training the Static Set on the first million queries of *Altavista* and we repeated the tests varying the Dynamic Set policy and the prefetching factor.

Looking at the curves in Figure 3.9, we can observe that the optimum value of f_{static} , i.e. the percentage of entries devoted to the Static Set, drop from 0.9 (training set composed by the first two-third of query log entries) to 0.4. The reason why this actually happens seems to be related to the *freshness* of the Static Set. The rationale is that the older the Static Set, the smaller the percentage of statically-locked entries actually used. This because the number of queries which continue to appear drop as the time goes on.

The estimation of how frequently the Static Set should be refreshed is an issue we reserve for a future work. We can use, for example the algorithm shown in [98].

3.6 Concurrent cache manager

The implementation of complex software systems should consider a number of aspects that rarely can coexists together. These aspects can be collected into two broad categories: *Software Quality*, and *System Efficiency*. Typically, the criteria used to evaluate the “*quality*” are: *modularity*, *flexibility*, and *maintainability*. While, when considering “*performance*”, one is more interested in optimizing the *response time*, the *scalability*, and the *throughput*.

In designing our caching system we tried to combine the two aspects of Software Quality and System Efficiency into a C++ implementation, which heavily relies on the use

of different Object Oriented (OO) techniques. For example, to improve system portability we used two multi-platform OO toolkits *STLPort* and *ACE*, which are highly optimized and thus ensure high performance. Moreover, in terms of system flexibility, we implemented our software by heavily using C++ features such as inheritance and templates. In particular, the use of templates permitted us to abstract from the specific data types being stored, and allows our system to be easily adapted to different formats. For instance, one can choose to store complete html pages of results, which usually contain URLs, snippets, titles, etc. along with formatting information (*Html cache*); otherwise, one can store only the numbers used in the WSE to identify the relevant documents corresponding to a given query (*DocID cache*).

As we anticipated in the introduction, our cache can easily be integrated into a typical WSE operating environment. A possible logical placement of our cache, as shown in Figure 3.1), is between the Web server and the mediator. When the mediator is responsible also for the preparation of the complete html page, the cache shown in the Figure should be a *DocID cache*. Otherwise, the cache can be a *Html cache* too.

In the following we will describe the modules composing our system. In particular, we will focus on the description of the data structures used to efficiently store and retrieve the elements contained into the cache. Our cache is fully associative, i.e. a query result may be associated with any entry of the cache. It is accessed through user queries, whose normalized format is (*keywords, page_no*). To ensure efficiency in the query lookup, we used a hash function which transforms the key (i.e., the user query) into the right table address. In case of cache hits we can thus perform this lookup-and-retrieve task in $O(1)$ time complexity.

The Static Set of SDC, hereinafter *StaticTable*, is stored as a simple associative memory that maps queries into their associated results. The *DynamicTable*, which is the dynamic section of the cache, not only implements the associative memory, but also maintains one of several orderings among the various cache entries. The specific orderings depend on the replacement policy implemented. When a cache miss occurs and no empty cache entries are available, these orderings are exploited to select the cache entry to be replaced. For example, if we adopt an *LRU* replacement policy, we have to select the least recently used cache entry, and thus the only ordering to maintain regards the recency of references. In practice, the *DynamicTable* implements these orderings through pointers used to interlink the cache entries, thus maintaining sorted lists. This avoids to copy the data entries of *CacheLookup* table when an ordering must be modified.

We designed our caching system to allow efficient concurrent accesses to its blocks. This is motivated by the fact that a WSE has to process several user queries concurrently. This is usually achieved by making each query processed by a distinct thread. The methods exported by our caching system are thus thread-safe and also ensure the mutual exclusion. In this regard, the advantage of SDC over a pure dynamic cache is related to the presence of the *StaticTable*, which is a read-only data structure. Multiple threads can thus concurrently lookup the *StaticTable* to search for the results of the submitted query. In case of a hit, the threads can also retrieve the associated page of results without synchronization. For this reason our caching system may sustain linear speed-up even in

configurations containing a very large number of threads. Conversely, the *DynamicTable* must be accessed in the critical section controlled by a mutex. Note, in fact, that the *DynamicTable* is a read-write data structure: while a cache miss obviously causes both the associative memory and relative list of pointers to be modified, also a cache hit entails the list pointers to be modified in order to sort the cache entries according to the replacement policy adopted.

Figure 3.10 shows the performance of our cache system in a multi-threading environment. In particular, the Figure plots, for different values of f_{static} , the scalability of the system as a function of the number of concurrent threads sharing a single cache. Scalability has been measured by considering the ratio between the wall-clock times spent by one and n threads to serve the same large bunch of queries. The replacement policy adopted in running the test was *LRU*, while the *speculative* prefetching factor was 5. In the case of a cache hit, the thread serving the query returns immediately the requested page of results, and gets another query. Conversely, when a query causes a cache miss, the thread sleeps for δ ms to simulate the latency of the WSE core in resolving the query. Then, it returns the page of results and begins to serve another query. In order to take into account the cost of prefetching additional results, the value δ was set according to the following formula: $\delta = 40\text{ms} + (20\text{ms} \cdot \log k)$, where k is the actual number of pages of results retrieved for the current query. Since in the test we used our *speculative* heuristic with a prefetching factor 5, δ resulted equal to 40 ms when the cache miss was caused by a query requiring the first page of results, and about 54 ms when the miss was served by prefetching also additional results.

In order to experimentally evaluate efficiency and efficacy of the proposed caching system, we have to simulate its inclusion between an http server and a WSE core query service (see Figure 3.1). In the tests conducted, the behavior of a multi-threaded WSE http server, which forwards user queries to the cache system and waits for query results, was simulated by actually reading the queries from a single log file. The WSE core query service, which is invoked to resolve those queries that have caused cache misses, was instead simply simulated by sleeping the process (or thread) which manages the query for a fixed amount of time.

As it can be seen, the system scales very well even when a large number of concurrent threads is exploited. The scalability of a purely static cache is super-optimal since the cache is accessed read-only, but high scalability values are achieved also when SDC is adopted. Note that even when a purely dynamic cache is adopted ($f_{static} = 0$), our system scales linearly with up to 250 concurrent threads: this is mainly due to the accurate software design. With a *SDC-LRU*_{0.8} cache of 50,000 blocks, no prefetching, average cache hit and miss management times are only $11\mu\text{s}$ and $36\mu\text{s}$, respectively (neglecting the WSE latency for misses).

3.7 Conclusions and Future works

In this chapter we presented a new policy for caching the query results of a WSE. The main enhancement over previous proposals regards the exploitation of past knowledge about queries submitted to the WSE to make more effective the management of our cache. In particular, since we noted that the most popular queries that are submitted to a WSE do not change very frequently, we maintain these queries and associated results in a read-only static section of our cache. The static section is updated at regular times on the basis of the WSE query log. Only the queries that cannot be satisfied by the static cache section compete for the use of a dynamic cache.

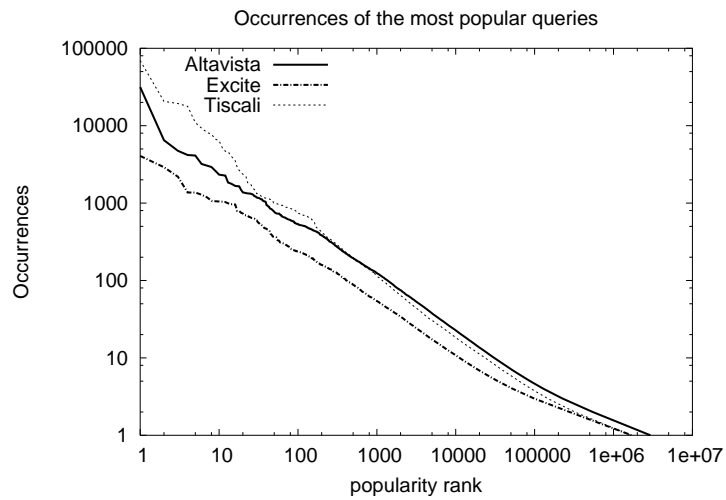
SDC, based on a static+dynamic cache, simplifies the choice of the replacement policy to adopt in the dynamic section of the cache. The presence of a static cache, which permanently stores the most frequently referenced pages, makes in fact *recency* the most important parameter to consider. As a consequence, a simple *LRU* policy handles effectively blocks replacement in our dynamic cache section, while some sophisticated (and often computationally expensive) policies specifically designed to exploit at the best both *frequency* and *recency* of references result less effective than *LRU* in the presence of a statically locked portion of the cache.

The benefits in adopting SDC were experimentally shown on the basis of tests conducted with three large query logs. In all the cases our strategy remarkably outperformed purely static or dynamic caching policies. We evaluated the hit-ratio achieved by varying the percentage of static blocks over the total, the size of the cache, as well as the replacement policy adopted for the dynamic section of our cache.

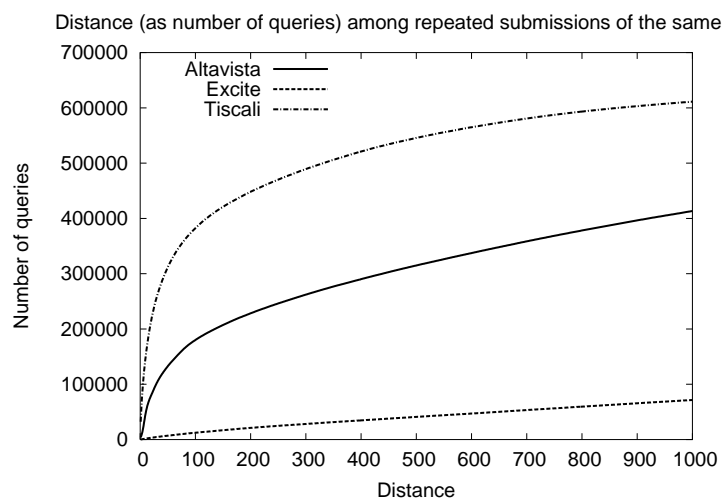
Moreover, we showed that WSE query logs also exhibit spatial locality. Users, in fact, often require subsequent pages of results for the same query. Our caching system takes advantage of this locality by exploiting a sort of *speculative* prefetching strategy which contemporary to a response to a user's request for the second page of results, formulates a request to the underlying WSE core for the results related to the following pages.

Finally, differently from other works, we evaluated cost and scalability of our cache implementation when executed in a multi-threaded environment. The SDC implementation resulted very efficient due to an accurate software design that allowed to make cache hit and miss times negligible, and to the presence of the read-only static cache that reduces the synchronization between multiple threads concurrently accessing the cache.

A future work regards the evaluation of a cache on the server-side subdivided into two parts: an *Html cache* at level L1 (placed on the machine that hosts http server), and a *DocID cache* at level L2 (placed on the machine that hosts the mediator). Moreover, we would like to evaluate the effectiveness of our technique for caching WSE query results at the proxy-side, on the basis of locally collected query logs. Since the users of a proxy generally belong to a homogeneous community (e.g. the departments of a University), we expect to find higher locality in the stream of queries submitted. Moreover, exploiting caching at this level both enhances user-perceived *QoS* and saves network bandwidth.



(a)



(b)

Figure 3.2: (a) Number of submissions of the most popular queries contained in the three query logs (log-log scale). (b) Distances (in number of queries) between subsequent submissions of the same query.

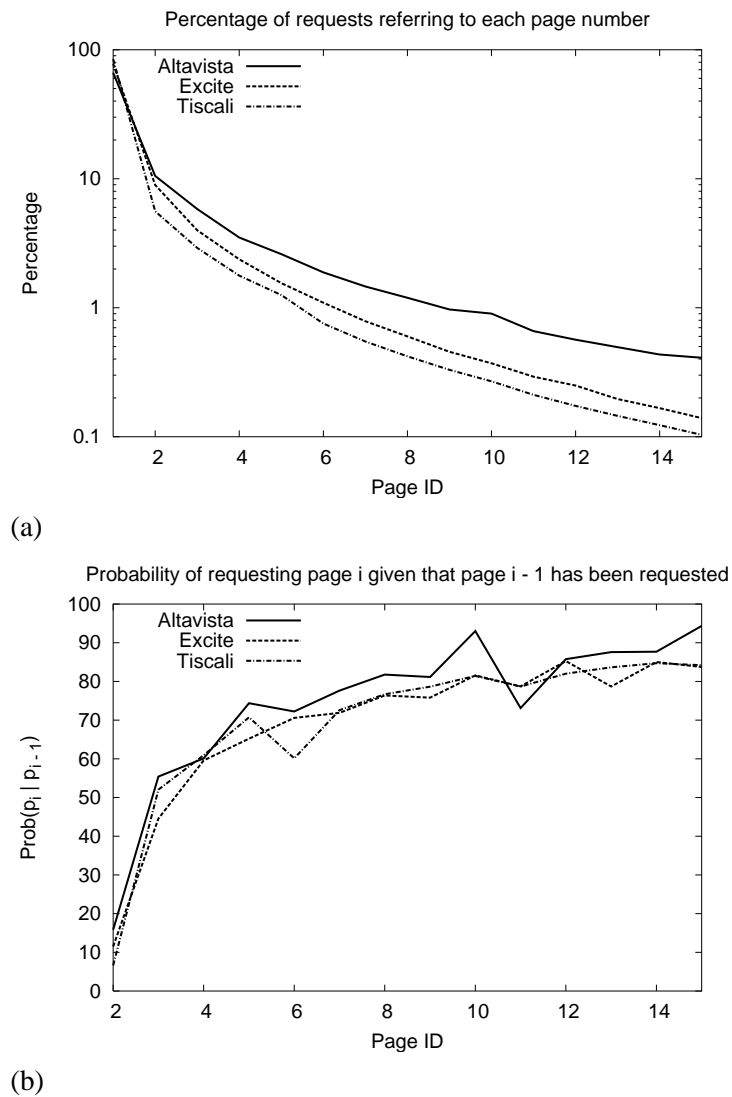


Figure 3.3: Spatial locality in the three query logs analyzed: (a) percentage of queries as a function of the index of the page requested; (b) probability of the occurrence of a request for the i -th page given a previous request for page $(i - 1)$.

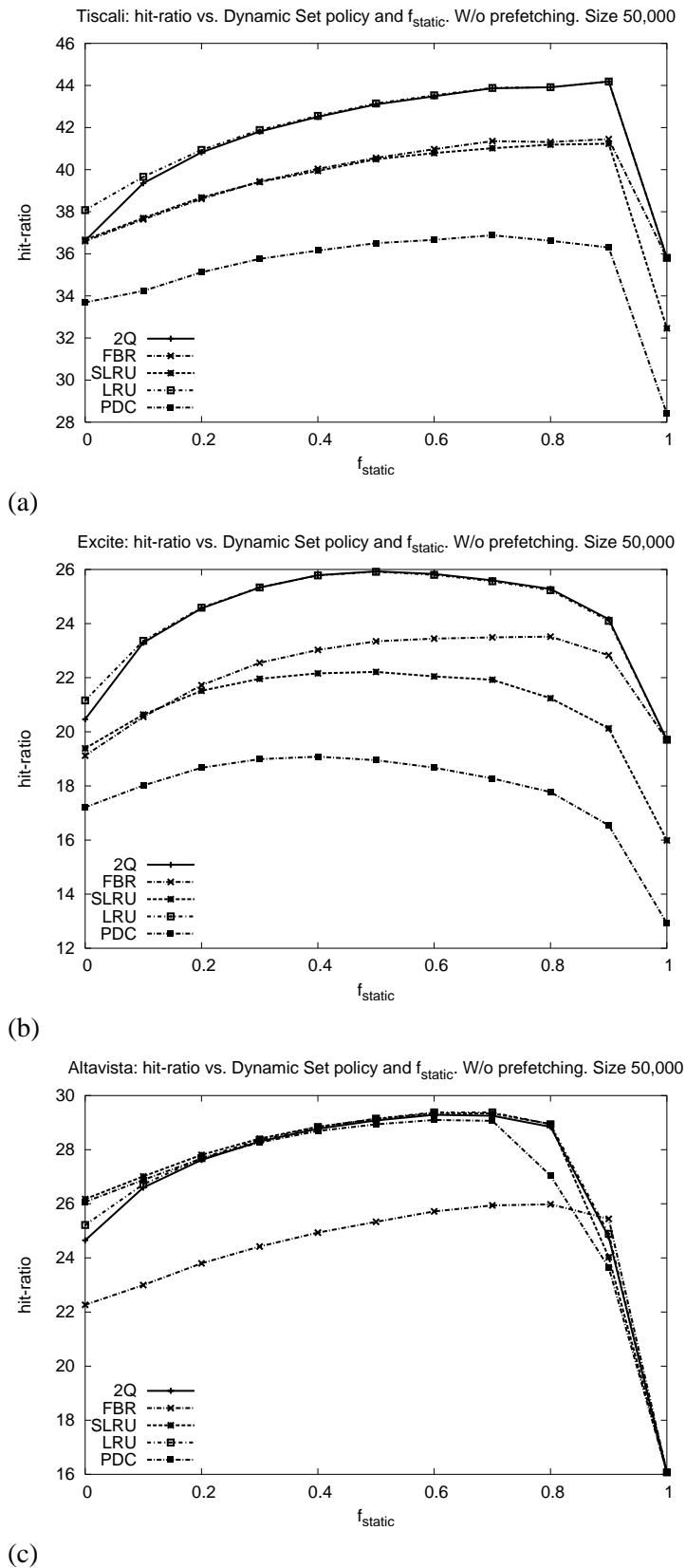
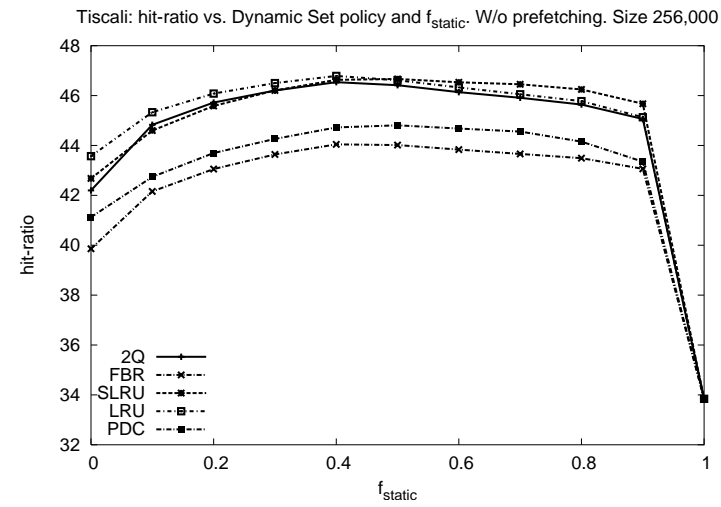
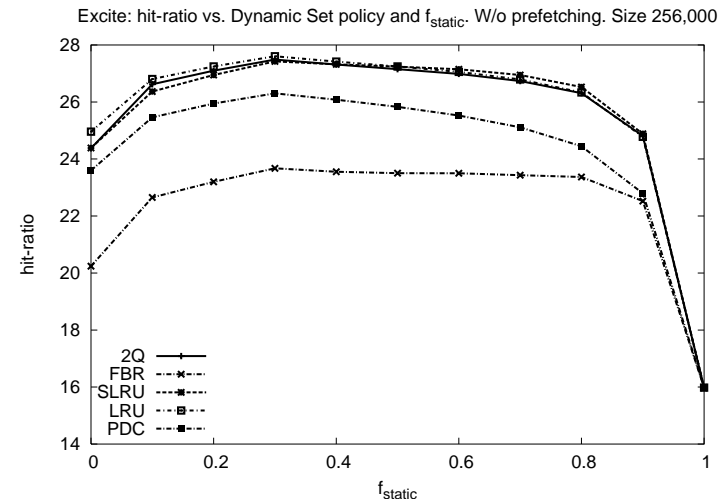


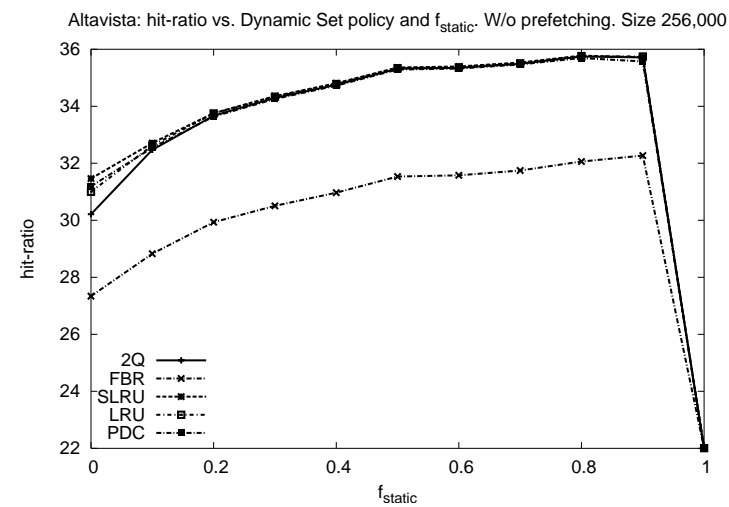
Figure 3.4: Hit-ratios obtained on the three query logs by using different replacement policies as a function of the ratio between static and dynamic cache entries: (a) *Tiscali*, (b) *Excite*, (c) *Altavista*. The tests performed are referred to a cache whose size was fixed to 50,000 entries.



(a)

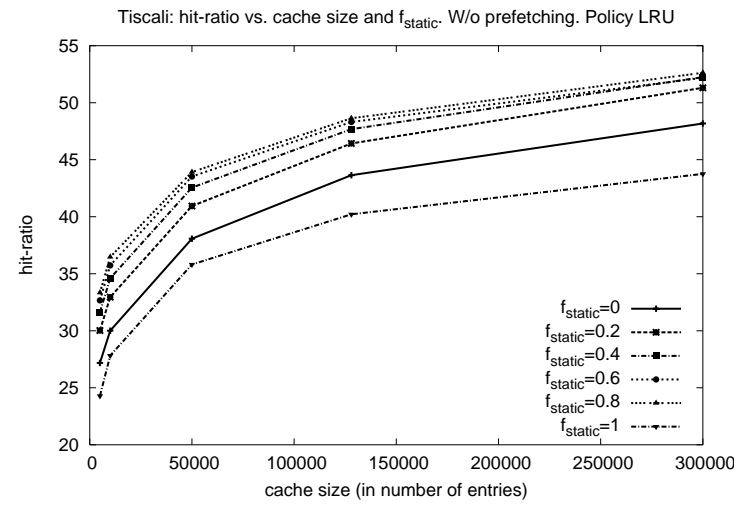


(b)

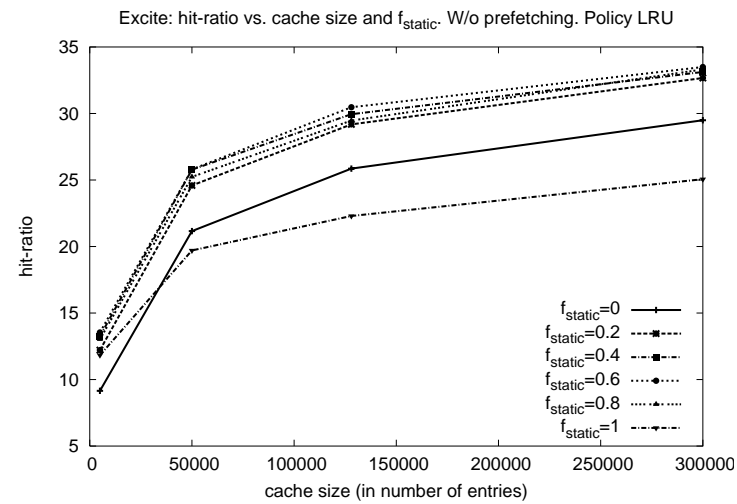


(c)

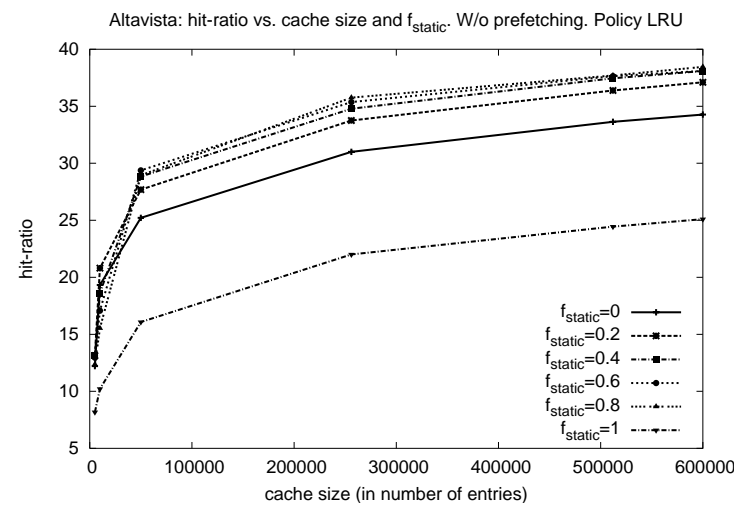
Figure 3.5: Hit-ratios obtained on the three query logs by using different replacement policies as a function of the ratio between static and dynamic cache entries: (a) *Tiscali*, (b) *Excite*, (c) *Altavista*. The tests performed are referred to a cache whose size was fixed to 256,000 entries.



(a)



(b)



(c)

Figure 3.6: Hit-ratios achieved on (a) *Tiscali*, (b) *Excite*, (c) *Altavista* logs as a function of the size of the cache and the f_{static} used.

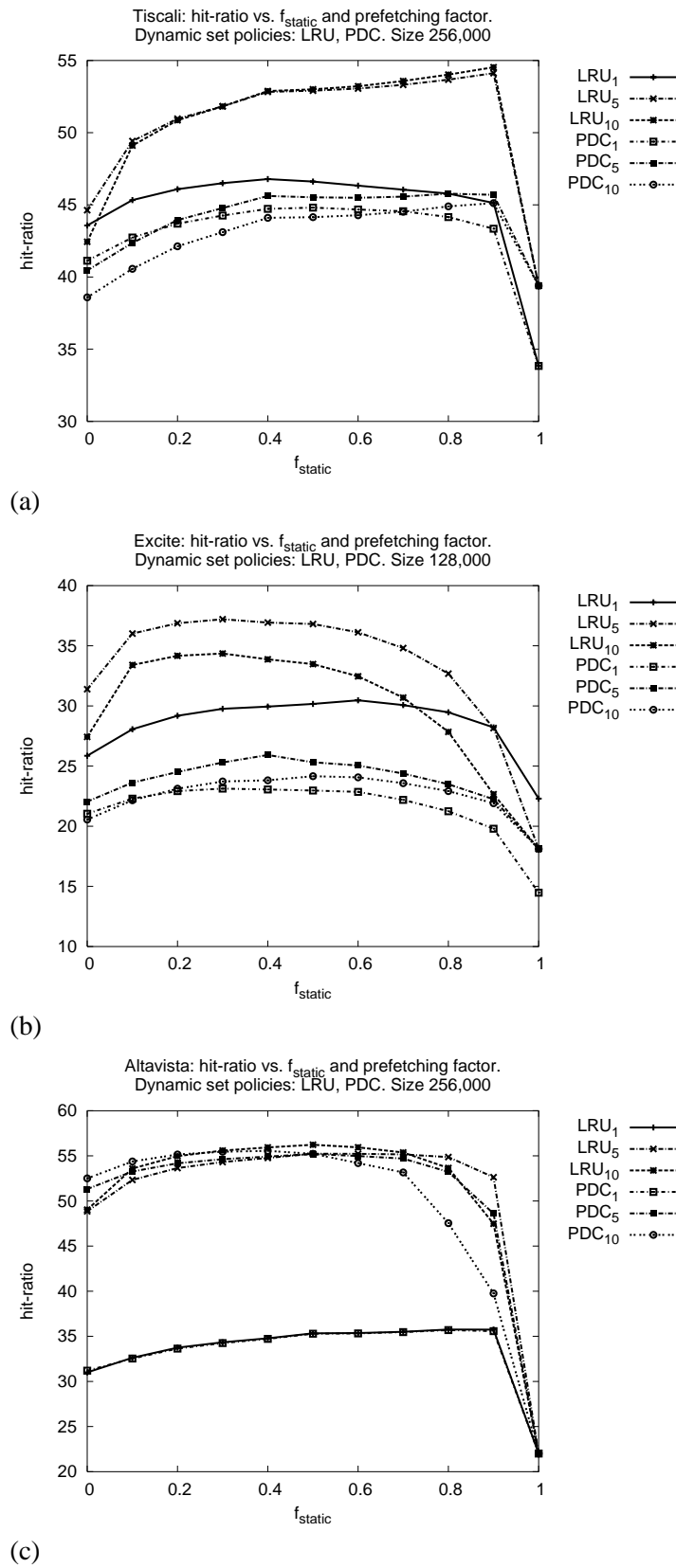


Figure 3.7: Hit-ratios obtained on the three query logs by using different replacement policies as a function of the prefetching factor: (a) *Tiscali*, (b) *Excite*, (c) *Altavista*. The tests performed are referred to a cache whose size was fixed to 256,000 entries for *Tiscali* and *Altavista*; 128,000 for *Excite*.

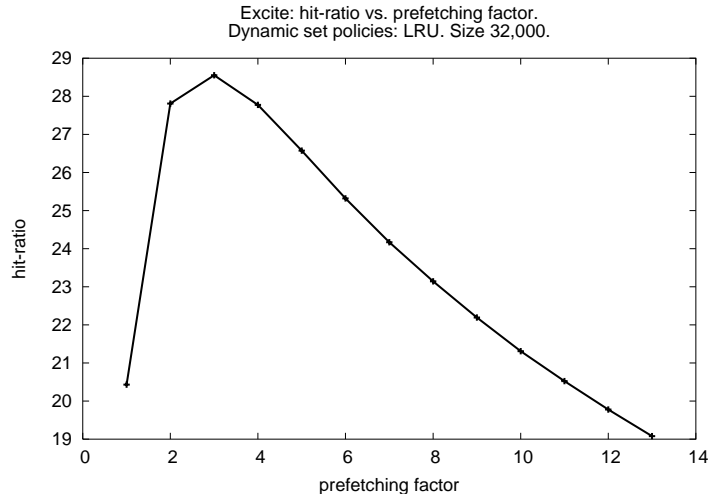


Figure 3.8: Hit-ratio variations vs. prefetching factor on the Excite query log with a cache of 32,000 elements and *LRU*.

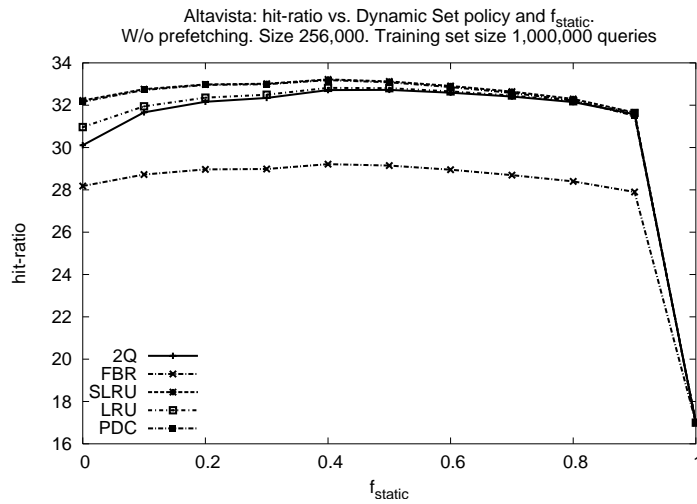


Figure 3.9: Hit-ratios obtained on the three query logs by using different replacement policies as a function of the ratio between static and dynamic cache entries on the *Altavista* query log. The tests performed are referred to a cache whose size was fixed to 256,000 entries. The training set in this tests is composed by the first million queries submitted.

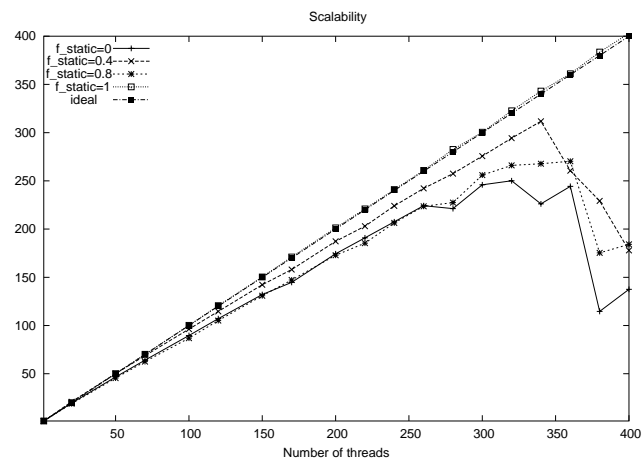


Figure 3.10: Scalability of our caching system for different values of f_{static} as a function of the number of concurrent threads used.

Chapter 4

Indexing Web Documents

Abstract

In the IR context, *indexing* is the process of developing a document representation by assigning content descriptors or terms to the documents of a collection. An *Indexer* is the module in a WSE system which is responsible to carry-out the indexing process. To process the ever-increasing volume of data while still providing acceptable completion times, we need to make use of scalable parallel IR algorithms. In this chapter we discuss the design of a parallel indexer for Web documents. By exploiting both data and pipeline parallelism, our prototype indexer efficiently builds a partitioned and compressed inverted index, a suitable data structure commonly utilized by modern Web Search Engines. We discuss implementation issues and report the results of preliminary tests conducted on a SMP PC.

Nowadays, Web Search Engines (WSEs) [11, 151, 20, 113] index hundreds of millions of documents retrieved from the Web. Parallel processing techniques can be exploited at various levels in order to efficiently manage this enormous amount of information. In particular it is important to make a WSE scalable with respect to the size of the data and the number of requests managed concurrently.

In a WSE we can identify three principal modules: the *Spider*, the *Indexer*, and the *Query Analyzer*. We can exploit parallelism in all the three modules. For the *Spider* we can use a set of parallel agents which visit the Web and gather all the documents of interest. Furthermore, parallelism can be exploited to enhance the performance of the *Indexer*, which is responsible for building an index data structure from the collection of gathered documents to support efficient search and retrieval over them. Finally, parallelism and distribution is crucial to improve the throughput of the *Query Analyzer* (see [113]), which is responsible for accepting user queries, searching the index for documents matching the query, and returning the most relevant references to these documents in an understandable form.

In this chapter we will analyze in depth the design of a parallel *Indexer*, discussing the realization and the performance of our WINGS (Web INdixinG System) prototype. While the design of parallel *Spiders* and parallel *Query Analyzers* has been studied in

depth, only a few papers discuss the parallel/distributed implementation of a Web Indexer [81, 102]. Consider that, when the collection is small and indexing is a rare activity, optimizing index-building is not as critical as optimizing run-time query processing and retrieval. However, with a Web-scale index, index building time also became a critical factor for two main reasons: *Scale and growth rate*, and *Rate of change* of the Web.

4.1 Indexing in Web Search Engines

Several sequential algorithms have been proposed, which try to well balance the use of core and out-of-core memory in order to deal with the large amount of input/output data involved. The *Inverted File (IF)* index [148] is the data structure typically adopted for indexing the Web. This is mainly due to two different reasons. The first is that an IF index allows the resolution of queries on huge collections of Web pages to be efficiently managed, and works very well for common Web queries, consisting of the conjunction of a few terms. Second, an IF index can be easily compressed to reduce the space occupancy in order to better exploit the memory hierarchy [132].

An IF index on a collection of Web pages consists of several interlinked components. The principal ones are: the *lexicon*, i.e. the list of all the *index terms* appearing in the collection, and the corresponding set of *inverted lists*, where each list is associated with a distinct term of the lexicon. Each inverted list contains, in turn, a set of *postings*. Each posting collects information about the *occurrences* of the corresponding term in the collection's documents. For the sake of simplicity, in the following discussion we will consider that each posting only includes the identifier of the document (*DocID*) where the term appears, even if postings actually store other information used for document ranking purposes (e.g. in our implementation each posting also includes the positions and the frequency of the term within the document, and context information like the appearance of the term within specific html tags).

Another important feature of the IF indexes is that they can be easily partitioned. In fact, let us consider a typical parallel Query Analyzer module: the index can be distributed across the different nodes of the underlying architecture in order to enhance the overall system's throughput (i.e. the number of queries answered per each second). For this purpose, two different partitioning strategies can be devised. The first approach requires to horizontally partition the whole inverted index with respect to the lexicon, so that each query server stores the inverted lists associated with only a subset of the index terms. This method is also known as *term partitioning* or *global inverted files*. The other approach, known as *document partitioning* or *local inverted files*, requires that each query server becomes responsible for a disjoint subset of the whole document collection (vertical partitioning of the inverted index). Following this last approach the construction of an IF index become a two-staged process. In the first stage each index partition is built locally and independently from a partition of the whole collection. The second phase is instead very simple, and is needed only to collect global statistics computed over the whole IF index.

Since the document partitioning approach provides better performance figures for processing typical Web queries than the term partitioning one, we adopted it in our Indexer prototype. Figure 4.1 illustrates this choice, where the document collection is here represented as a set of html pages.

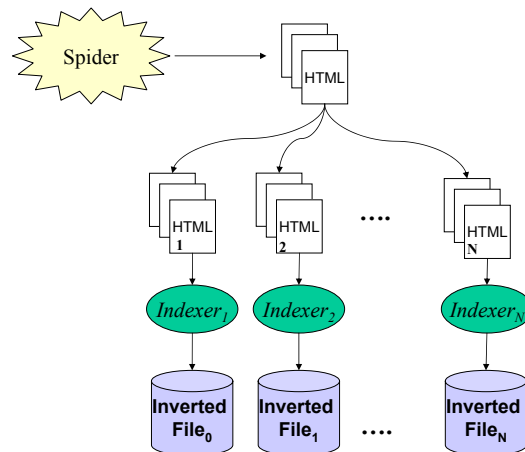


Figure 4.1: Construction of a distributed index based on the document partition paradigm, according to which each local inverted index only refers to a partition of the whole document collection.

Note that in our previous work [113] we conducted experiments where we pointed out that the huge sizes of the Web make the global statistics useless. Just to give a hint of the reasons why this is true remember that word occurrences within documents follow a Zipfian distribution [158]¹. If we consider a generic collection we uniformly select a subset of documents, then it is very likely that, due to the self-similarity characteristics of Zipf's law [4], the subcollection will maintain the same distribution of word occurrences of the global collection. In particular, the local statistic regarding the frequency of occurrence of words within documents (*idf*) will be approximately the same than the global one. For this reason, we did not consider this phase in the design of our indexer.

4.2 A pipelined indexing system

The design of a parallel Indexer for a WSE adopting the document partition approach (see Figure 4.1), can easily exploit *data parallelism*, thus independently indexing disjoint sub-collections of documents in parallel. Besides this natural form of parallelism, in this chapter we want to study in depth the parallelization opportunities within each instance of the Indexer, say *Indexer_i*, which accomplishes its indexing task on a disjoint partition *i* of the whole collection.

¹At least this is true for English texts.

Document Id	Document Text
1	Pease porridge hot, pease porridge cold.
2	Pease porridge in the pot.
3	Nine days old.
4	Some like hot, some lite it cold.
5	Some like in the pot.
6	Nine days old.

(a)

Term	Postings list
cold	1, 1
days	3, 6
hot	1, 4
in	2, 5
it	4, 5
like	4, 5
nine	3, 6
old	3
...	...

(b)

Table 4.1: A toy text collection (a), where each row corresponds to a distinct document, and (b), the corresponding inverted index, where the first column represents the lexicon, while the last column contains the inverted lists associated with the index terms.

The job performed by each *Indexer_i* to produce a local inverted index is apparently simple. If we consider the collection of documents as modeled by a matrix (see Table 4.1.(a)), building the inverted index simply corresponds to transpose the matrix (see Table 4.1.(b)). This matrix transposition or inversion can be easily accomplished in-memory for small collections. Unfortunately, a naive in-core algorithm becomes rapidly unusable as the size of the document collection grows. Note that, with respect to a collection of some GBs of data, the size of the final lexicon is usually a few MBs so that it can be maintained in-core, while the inverted lists cannot fit into the main memory and have to be stored on disk, even if they have been compressed.

To efficiently index large collections, a more complex process is required. The most efficient techniques proposed in literature [148], are all based on external memory sorting algorithms. As the document collection is processed, the Indexer associates a distinct *DocID* with each document, and stores into a in-core buffer all the pairs $\langle Term, DocID \rangle$, where *Term* appears at least once in document *DocID*. Buffer size occupies as much memory as possible, and when it becomes full, it is sorted by increasing *Term* and by increasing *DocID*. The resulting *sortedruns* of pairs are then written into a temporary file on disk, and the process repeated until all the documents in the collection are processed. At the end of this first step, we have on disk a set of sorted runs stored into distinct files. We can thus perform a multi-way merge of all the sorted runs in order to materialize the final inverted index.

According to this approach, each *Indexer_i* works as follows: it receives a stream of

query. For the sake of clarity, we will omit all these details in the following discussion.

The latter module of the first pipeline, $Inverter^{pre}$, thus receives from the *Parser* stage a stream of terms associated with distinct $DocIDs$, incrementally builds a lexicon by associating a $TermIDs$ with each distinct term, and stores on disk large sorted runs of pairs $\langle TermID, DocID \rangle$. Before storing the run, it has to order them first by $TermID$, and then by $DocID$. Note that the use of integer $TermID$ and $DocID$ not only reduces the size of each run, but also makes faster comparisons and thus runs' sorting. $Inverter^{pre}$ has a sophisticated main memory management, since the lexicon has to be kept in-core, each run has to be as large as possible before flushing to disk, and we have to avoid memory swapping.

When the first pipeline $Indexer^{pre}$ ends processing a given document partition, the second pipeline $Indexer^{post}$ can start its work. The input to this second pipeline is exactly the output of $Indexer^{pre}$, i.e., the set of sorted runs and the lexicon relative to the document partition. The first stage of the second pipeline is the $Inverter^{post}$ module, whose job is to produce a single sorted run starting from the various disk-stored sorted runs. The sorting algorithm is very simple: it is a in-core multi-way merge of the n runs, obtained by reading into the main memory the first block b of each run, where the size of the block is carefully chosen on the basis of the memory available. The top pairs of all the blocks are then inserted in a (min-)heap data structure, so that the top of the heap contains the smallest $TermID$, in turn associated with the smallest $DocID$. As soon as the lowest pair p is extracted from the heap, another pair, coming from the same sorted run containing p (i.e., from the corresponding block), is inserted into the heap. Finally, when an in-core block b is completely emptied, another block is loaded from the same disk-stored run. The process clearly ends when all the disk-stored sorted runs have been entirely processed, and all the pairs extracted from the top position of the heap.

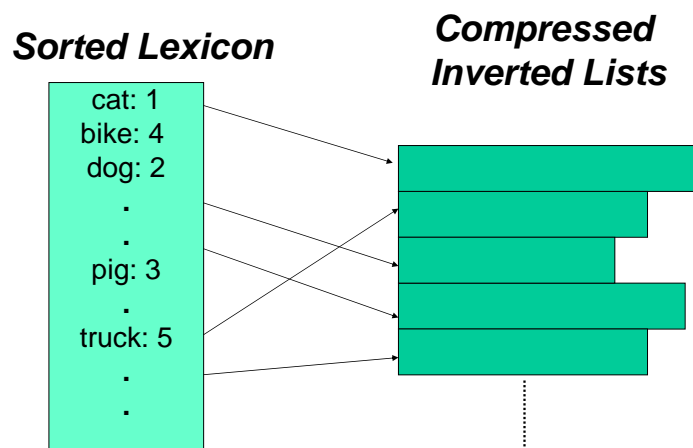


Figure 4.3: Inverted file produced by the *Flusher*.

Note that as soon as $Indexer^{post}$ extracts each ordered pair from the heap, forwards it to the *Flusher*, i.e. the latter stage of the second pipeline. This stage receives in order all the postings associated with each $TermID$, compresses them by using the usual techniques based on the representation of each inverted list as a sequence of gaps between sorted $DocIDs$, and stores each compressed list on disk. A scratch of the inverted index produced is shown in Figure 4.3. Note that the various lists are stored in the inverted file according to the ordering given by the $TermID$ identifiers, while the lexicon shown in figure has to be finally sorted according to lexicographic order given by the corresponding terms.

4.2.1 Experimental results

All the tests were conducted on a 2-way SMP equipped with two Intel 2.0 GHz Xeon processors, one GB of main memory, and one 80 GB IDE disk.

For the communication between the various stages of the *Indexer* pipelines, we used an abstract communication layer that can be derived in order to exploit several mechanisms. In the tests we performed, since the pipeline stages are mapped on the same node, we exploited the System V message queue IPC.

In order to evaluate the opportunity offered by the pipelined parallelization scheme illustrated above, we have first evaluated the computational cost of the *Parser*, the *Inverter*, and the *Flusher* module, where each of them uses the disk for input/output.

Collection Size (GB)	Time (s) <i>Parser</i>	Time (s) <i>Inverter</i>	Time (s) <i>Flusher</i>	Tot. Throughput (GB/h)
1	3610	1262	144	0.71
2	7044	3188	285	0.68
3	11355	4286	429	0.67
4	14438	5714	571	0.69
5	18047	6345	725	0.71

Table 4.2: Sequential execution times for different sizes of the document collection.

As you can observe from the execution times reported in Table 4.2, the most expensive module is the *Parser*, while the execution time of the *Flusher* is, on average, one order of magnitude smaller than those of the *Inverter*. Moreover, we have to consider that, in the pipeline version, the *Inverter* module will be split into two smaller ones, $Inverter^{pre}$ and $Inverter^{post}$, whose cost is smaller than the whole *Inverter*. From the above considerations, we can conclude that the pipeline implementation will result in an *unbalanced computation*, so that if we execute a single pipelined instance $Indexer_i$ on a 2-way multiprocessors, processors, it should result in a under-utilization of the workstation. In particular, the $Inverted^{pre}$ should waste most of its time waiting for data coming from the *Parser*.

When a single pipelined *Indexer_i* is executed on a multiprocessor, a way to increase the utilization of the platform is to try to balance the throughput of the various stages. From the previous remarks, in order to balance the load we could increase the throughput of the *Parser*, i.e. the most expensive pipeline stage, for example by using a multi-thread implementation where each thread independently parses a distinct document³. The other way to improve the multiprocessor utilization is to map multiple pipelined instances of the indexer, each producing a local inverted index from a distinct document partition.

Coll. Size (GB)	No. of instances			
	1	2	3	4
1	1.50	2.01	2.46	2.57
2	1.33	2.04	2.44	2.58
3	1.38	1.99	2.38	2.61
4	1.34	2.04	2.45	2.64
5	1.41	2.05	2.45	2.69

Table 4.3: Total throughput (GB/s) when multiple instances of the pipelined *Indexer_i* are executed on the same 2-way multiprocessor.

In this chapter we will evaluate the latter alternative, while the former one will be the subject of a future work. Note that when we execute multiple pipeline instances of *Indexer_i*, we have to carefully evaluate the impact on the shared multiprocessor resources, in particular the disk and the main memory. As regards the disk, we have evaluated that the most expensive stage, i.e. the *Parser*, is compute-bound, so that the single disk suffices to serve requests coming from multiple *Parser* instances. As regards the main memory, we have tuned the memory management of the stages *Inverter^{pre}* and *Inverter^{post}*, which in principle could need the largest amount of main memory to create and store the lexicon, store and sort the runs before flushing to the disk, and to perform the multi-way merge from the sorted runs.

In particular, we have observed that we can profitably map up to 4 instances of distinct pipelined *Indexers_i* on the same 2-way processor, achieving a maximum throughput of 2.64 GB/hour. The results of these tests are illustrated in Table 4.3. Note that, when a single pipelined *Indexer_i* is executed, we were however able to obtain a optimal speedup over the sequential version ($\simeq 1.4$ GB/h vs. $\simeq 0.7$ GB/h), even if the pipeline stages are not balanced. This is due to the less expensive in-core pipelined data transfer (based on message queues) of the pipeline version, while the sequential version must exploit the disk to save intermediate results.

³The *Parser* is the only pipeline stage that can be further parallelized by adopting a simple data parallel scheme.

4.3 Results assessment and future works

We have discussed the design of WINGS, a parallel indexer for Web contents that produces local inverted indexes, the most commonly adopted organization for the index of a large-scale parallel/distributed WSE. WINGS exploits two different levels of parallelism. Data parallelism, due to the possibility of independently building separate inverted indexes from disjoint document partitions. This is possible because WSEs can efficiently process queries by broadcasting them to several searchers, each associated with a distinct local index, and by merging the results. In addition, we have also shown how a limited pipeline parallelism can be exploited within each instance of the indexer, and that a low-cost 2-way workstation equipped with an inexpensive IDE disk is able to achieve a throughput of about $2.7 \text{ GB}/\text{hour}$, when processing four document collections to produce distinct local inverted indexes. Further work is required to assess the performance of our indexing system on larger collections of documents, and to fully integrate it within our parallel and distributed WSE prototype [113]. Moreover, we plan to study how WINGS can be extended in order to exploit the inverted lists active compression strategy discussed in [132].

Chapter 5

Index Compression Issues

Abstract

Web Search Engines provide a large-scale text document retrieval service by processing huge *Inverted File* indexes. Inverted File indexes allow fast query resolution and good memory utilization since their d -gaps representation can be effectively and efficiently compressed by using variable length encoding methods. This chapter proposes and evaluates some algorithms aimed to find an assignment of the document identifiers which minimizes the average values of d -gaps, thus enhancing the effectiveness of traditional compression methods. We ran several tests over the Google contest collection in order to validate the techniques proposed. The experiments demonstrated the scalability and effectiveness of our algorithms. Using the proposed algorithms, we were able to sensibly improve (up to 20.81%) the compression ratios of several encoding schemes.

Compressing the huge index of a Web Search Engine (WSE) entails a better utilization of memory hierarchies and thus a lower query processing time [126]. During the last years several works addressed the problem of index compression. The majority of them focused on devising effective and efficient methods to encode the document identifiers (*DocIDs*) contained in the posting lists of Inverted File (IF) indexes [126, 105, 7, 6, 151]. Since posting lists are ordered sequences of integer DocID values, and are usually accessed by scanning them from the beginning, these lists are stored as sequences of d -gaps, i.e. differences between successive DocID values. d -gap lists are then compressed by using variable-length encodings, thus representing smaller integers in less space than larger ones. Variable-length encoding schemes can be **bitwise**, or **bytewise**:

- In bitwise schemes, the list of integers is stored as a sequence of variable-length codewords, each composed of a variable number of bits. Well-known bitwise schemes include: *Elias' gamma*, *Delta*, *Golomb-Rice* [151], and *Binary Interpolative* coding [105]. Bitwise codes, in general, achieve very good compression ratios. The main of these methods is the relatively high decoding time, which may negatively impact on the query processing performance of the system. To overcome

this drawback, Anh and Moffat recently proposed a very effective bitwise encoding schema, which enhances considerably the decoding performance [6].

- In bitwise encoding, each integer is represented using a fixed and integral number of bytes. In its simplest form, the seven least significant bits of each byte are used to encode an integer, while the most significant bit is used as a sort of “continuation bit”, to indicate the existence of following bytes in the representation of the integer. An effective bitwise method, where word-alignment is retained, even at the cost of some bits wasted within each word, has been recently proposed by Anh and Moffat [7]. Bitwise codes have low decoding time, but are, in general, less effective than bitwise ones.

Since small d -gaps are much more frequent than large ones within postings lists, such variable-length encoding schemes allow IF indexes to be represented concisely. This feature of posting lists is called *Clustering property*, and is passively exploited by compression algorithms. However, by permuting DocIDs in a way that increases the frequency of small d -gaps, we may likely enhance the effectiveness of any variable-length encoding schema. Other works previously addressed this possibility [128, 15, 132, 133].

5.1 Compression of Inverted File

IF is the most widely adopted format for the index of a WSE due to its relatively small footprint and the efficiency in resolving keyword-based queries [151]. An IF, basically, follows the concept of the index of a book. For a given document corpora, an IF index maps each distinct term to a posting list of references (i.e. *docIDs*) to all the documents containing the term itself. Further, each entry also stores other information useful for ranking the relevance of a document with respect to a user’s query. For instance: the number of occurrences, the position of each occurrence of the term within the document, etc.

In this work, anyway, we concentrate our efforts toward finding efficient algorithms for assigning the docIDs in order to enhance the compressibility of the index. Thus we simplify the model by just considering a posting list composed by lists of integer numbers representing the docIDs for the document collection considered. Figure 5.1 shows a fragment of an Inverted File in our “*reduced*” model.

Since each posting list is usually sorted by docID¹, and usually a list is accessed by scanning its elements from the beginning, a posting list is usually stored using a *difference coding* technique [126, 105, 151]. The idea of difference coding can be illustrated by a simple example. Let us consider the posting list $((apple; 5)1, 4, 10, 20, 23)$ indicating that the term *apple* occurs in 5 documents: 1, 4, 10, 20, and 23. Using a naïve representation we could store each id using a fixed number of bit. This approach would require

¹The posting list are sorted to enable efficient resolution of boolean queries [148].

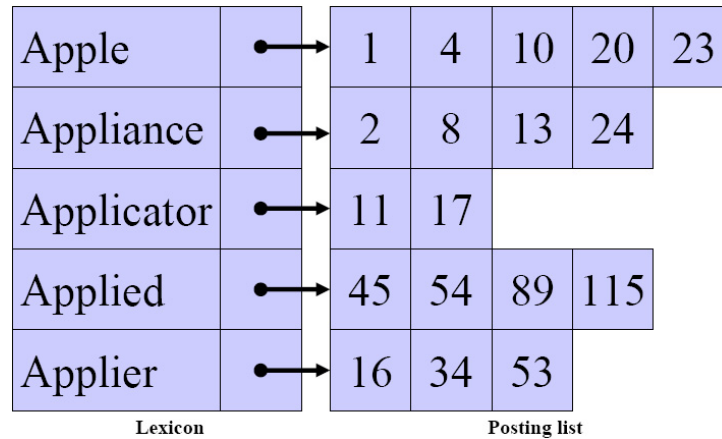


Figure 5.1: The structure of an Inverted File in our "reduced" model.

$O(\log(n))$ bits for each entry of the list (where n is the number of documents in the collection considered). Nevertheless, exploiting some structural properties of the lists we could reduce the space occupancy. We can, in fact, rewrite the list as $((apple; 5)1, 3, 6, 10, 3)$ storing the so called *d_gaps*, i.e. the differences between successive identifiers, instead of storing the actual docIDs. Since *d_gaps* are smaller than docIDs, they can be effectively encoded by a prefix-free code.

Several methods have been proposed during these years: *Gamma*, *Delta*, and *Golomb* codes, just to name a few [50, 65, 103, 110, 126, 105, 151]. The common feature of these codes is their variable-length representation: smaller numbers are more "shrunked" than larger ones. Indeed, in general we can consider two macro-classes of compression scheme:

1. *variable-bit*, also known as *bitwise*, in which each posting is stored using a bit-aligned code whose length depends on the encoded integer;
2. *variable-byte* (i.e. *bytewise*), in which each posting is stored employing a byte-aligned code.

5.1.1 Bitwise encoding.

Well-known bitwise schemes include: *Elias' gamma*, and *delta* [50], *Golomb-Rice* coding [65], *Binary Interpolative* coding [105].

Elias' gamma coding [50] is a bitwise method which stores a positive integer k by $1 + \lfloor \log_2 k \rfloor$ stored as a unary code, followed by the binary representation of k without its most significant bit.

As an example let us consider the encoding of the number 9. Using *gamma* 9 is represented by 1110001, since: $1 + \lfloor \log_2 9 \rfloor = 4$ or 1110 in unary, and 9 is 001 in binary with the most significant bit removed. In this way, 1 is represented by 1, that is, is represented

in just one bit. To decode the number we count the number of 1s in the unary part of the encoding (3 in the example) and we get the same number of following bits (skipping the end marker 0) yielding to 1001 after we reinsert the most significant bit.

Obviously this code is inefficient for storing large integers. In [150] the authors showed that gamma coding is relatively inefficient for storing integers larger than 15.

Elias' delta codes, instead, are suitable if we want to store large integers, but is very inefficient for small ones. For an integer k , a *delta* code stores the *gamma* code representation of $1 + \lfloor \log_2 k \rfloor$ and then the binary representation of k without its most significant bit.

Let us consider again the number 9. Using *delta*, 9 is represented by 11000001. In fact 11000 is the *gamma* representation of $1 + \lfloor \log_2 9 \rfloor = 4$ while, again, 001 is 9 in binary without the most significant bit. As expected the length of this encoding is greater than *gamma* for 9. On the other hand, let us consider the number 50. Using *gamma* we obtain 11111010010, while using *delta* we obtain 1101010010 that is we save 1 bit of space.

Golomb-Rice encoding [65] has been shown to offer more compact storage of integers and faster retrieval than the Elias codes [150]. Golomb codes are built under the basic assumption that the integers to be stored are distributed randomly. Coding of an integer k using Golomb code with respect to a parameter b is as follows. The code that is emitted is in two parts: first, the unary code of a quotient q is emitted, where $q = \lfloor (k - 1) / b \rfloor + 1$; second, a binary code is emitted for the remainder r , where $r = k - q \times b - 1$, this code can require either $\lfloor \log_2 b \rfloor$, or $\lceil \log_2 b \rceil$ bits to be represented. The value for the parameter b should be estimated. In [151] has shown that if b satisfies $(1 - p)^b + (1 - p)^{b+1} \leq 1 < (1 - p)^b + (1 - p)^{b-1}$, then this method generates an optimal prefix free code for a geometric distribution. Rice coding is a variant of Golomb, where b can only assume values that are powers of two. In this way the remainder r will always require exactly $\lceil \log_2 b \rceil$ bits to be stored.

Binary Interpolative encoding [105] exploits the so called *Clustering* property of word appearances. Basically, the clustering property states that word occurrences are not uniformly distributed across all the documents. Instead, they are clustered forming several runs of d-gaps equal to 1.

The Binary Interpolative method can be easily explained by means of a simple example. Let us assume we know the complete inverted file entry. For instance, let us consider the inverted list for the term t contained in seven different documents identified by the numerical values 3, 8, 9, 11, 12, 13, 17: $(t; 7; 3, 8, 9, 11, 12, 13, 17)$. The coding method recursively halves the range and calculates the lowest possible value and highest possible value for the number in the middle. So for the example above, we first look at the 4th number 11, then 8, which is the middle number in the left three documents, then 3, 9, then 13 (the middle number in the right three documents), and finally 12, 17. We use $(x; lo; hi)$ to denote that x lies in the interval $[lo; hi]$. Then for the example above, we can get the following sequence: $(11, 4, 17)$ since 11 is the 4th number among the seven documents, and there are twenty documents in total.

$(8, 2, 9); (3, 1, 7); (9, 9, 10); (13, 13, 19); (12, 12, 12)$.

Interpolative code, in general, can achieve the best compression ratio, but it requires a

longer decoding time than the other methods we have described.

5.1.2 Bytewise encoding.

In bitwise coding an integer is stored using a fixed and integral number of bytes. In its simplest form, the least seven bits of each byte are used to store the value of the integer, while the most significant bit is equal to zero if no additional blocks follow, otherwise it will be set to one. As an example see Figure 5.2.

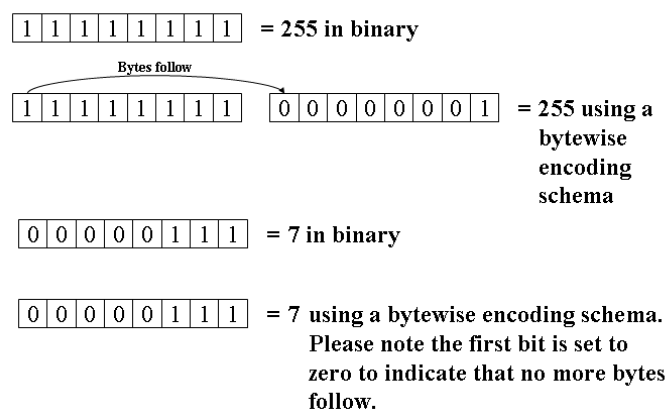


Figure 5.2: Two examples of integer encoded following a variable-byte code.

As already mentioned in the introduction, data compression in general can improve the price/performance ratio of computing systems by allowing the storing of large data in the faster levels of the memory hierarchy [139].

To demonstrate this fact in the case of Inverted File Indices, in [126] the authors investigate the performance of different index compression schemes through experiments on large query sets and collections of Web Documents. Among the main results discussed in the paper, the authors demonstrate that they define surprising:

- For a 20 gigabyte collection, where the index is several times larger than the main memory, optimized bitwise schema more than halve the average query response time compared to the fastest bitwise approach. This is due to the fact that in bitwise compression schema all the codewords are byte-aligned and thus retrieving a codeword doesn't involve expensive shift-mask operations.
- For a small collection, where the uncompressed index fits in main memory, a bitwise compressed index can still be processed faster than the uncompressed one.

Obviously the disadvantage of bytewise compressed indexes is that they are up to 30% larger than the bitwise compressed counterparts.

Shieh *et al.* [128] proposed a DocID reassignment algorithm adopting a Traveling Salesman Problem (TSP) heuristic. A *similarity* graph is built by considering each document of the collection as a vertex, and by inserting an edge between any pair of vertexes whose associated documents share at least one term. Moreover, edges are weighted by the number of terms shared by the two documents. The TSP heuristic algorithm is then used to find a cycle in the *similarity* graph having maximal weight and traversing each vertex exactly once. The suboptimal cycle found is finally broken at some point, and the DocIDs are reassigned to the documents according to the ordering established. The rationale is that since the cycle preferably traverses edges connecting documents sharing a lot of terms, if we assign close DocIDs to these documents, we should expect a reduction in the average value of *d*-gaps, and thus in the size of the compressed IF index. The experiments conducted demonstrated a good improvement in the compression ratio achieved. Unfortunately, this technique requires to store the whole graph in the main memory, and is too expensive to be used for real Web collections: the authors reported that reordering a collection of approximately 132,000 documents required about 23 hours and 2.17 GBytes of main memory.

Also Blelloch and Blandford [15] proposed an algorithm (hereinafter called *B&B*) that permutes the document identifiers in order to enhance the clustering property of posting lists. Starting from a previously built IF index, a similarity graph G is considered where the vertexes correspond to documents, and the edges are weighted with the *cosine similarity* [21] measure between each pair of documents. The *B&B* algorithm recursively splits G into smaller subgraphs $G_{l,i} = (V_{l,i}, E_{l,i})$ (where l is the level, and i is the position of the subgraph within the level), representing smaller subsets of the collection. Recursive splitting proceeds until all subgraphs become singleton. The DocIDs are then reassigned according to a *depth-first* visit of the resulting tree. The main drawback of this approach is its high cost both in time and space: similarly to [128] it requires to store the whole graph G in the main memory. Moreover, the graph splitting operation is expensive, although the authors proposed some effective sampling heuristics aimed to reduce its cost. In [15] the results of experiments conducted with the TREC-8 ad hoc track collection are reported. The enhancement of the compression ratio obtained is significant, but execution times reported refer to tests conducted on a sub-collection of only 32,000 documents. The paper addresses relevant issues, but due to its cost, also the *B&B* algorithm seems unfeasible for real Web collections.

In our opinion, another drawback of the previous approaches is that they focus on *reassigning* DocIDs appearing in a previously built IF index. The innovative point of our work is a bunch of DocID assignment techniques according to which DocIDs are assigned on the fly, during (and not after) the inversion of the document collection. In order to compute efficiently and effectively a good assignment, a new model to represent the collection of documents is needed. We propose a model that allows the assignment algorithm to be placed into the typical spidering-indexing life cycle of a WSE. Our model, hereinafter called *Transactional Model*, is based on the popular *bag-of-words* model, but

it does not consider the *within-doc* frequency of the terms. In a previous work [132], we presented preliminary results relative to one of the algorithms discussed in this chapter. Here we extend and complete the work by proposing and comparing several scalable and space-effective algorithms that can be used to assign DocIDs while the spidered collection is being processed by the Indexer. This means that when the index is actually committed on the disk, the new DocID assignment has been already computed. Conversely, the other methods proposed so far require that the IF index has already been computed before.

5.2 The Assignment Problem

Let $D = \{d_1, d_2, \dots, d_{|D|}\}$ be a set of $|D|$ textual documents. Moreover, let T be the set of distinct terms $t_i, i = 1, \dots, |T|$, present in D . Let G be a bipartite graph $G = (V, E)$, where the set of vertexes $V = T \cup D$ and the set of edges E contains arcs of the form $(t, d), t \in T$ and $d \in D$. An arc (t, d) appears in E if and only if term t is contained in document d .

Definition 1. A document assignment for a collection of documents D is defined as a bijective function π :

$$\pi : D \rightarrow \{1, \dots, |D|\}$$

that maps each document d_i into a distinct integer identifier $\pi(d_i)$.

Definition 2. Let l_i^π be the posting list associated with a term t_i . This list refers to both the set of vertexes $d_j \in D$ making up the neighborhood of vertex $t_i \in T$, and a given assignment function π :

$$l_i^\pi = \langle \pi(d_j) \mid (t_i, d_j) \in E \rangle \quad i = 1..|T|$$

The posting list is ordered. More formally, if $l_{i,u}^\pi$ and $l_{i,v}^\pi$ are respectively the u -th and v -th elements of l_i^π , then $l_{i,u}^\pi < l_{i,v}^\pi$ iff $u < v$.

The compression methods commonly used to encode the various posting lists l_i^π exploit a *dgap*-based, representation of lists. Before encoding l_i^π , the list is thus transformed into a list of dgaps of the form $(l_{i,k+1}^\pi - l_{i,k}^\pi)$, i.e., gaps between successive document identifiers. Let \bar{l}_i^π be the dgap-based representation of l_i^π .

Definition 3. Let L^π be the set of all $\bar{l}_i^\pi, i = 1..|T|$ making up an IF index. We can define the size of the IF index encoded with m as:

$$PSize_{L^\pi}^m = \sum_{i=1, \dots, |T|} Encode_m(\bar{l}_i^\pi)$$

where $Encode_m$ is a function that returns the number of bits required to encode the list \bar{l}_i^π .

Definition 4. The document assignment problem is an optimization problem, which aims to find the assignment π that yields the most compressible IF index with a given method m :

$$\min_{\pi} PSize_{L^\pi}^m$$

No. of documents	Bits per Postings						Assignment time in seconds
	Random		<i>B&B</i>		Gain(%)		
	Interp.	Var. Byte	Interp.	Var. Byte	Interp.	Var. Byte	
81,875	6.53	9.67	5.31	9.17	18.68	5.17	219.84
163,822	6.56	9.70	5.26	9.24	19.82	4.74	492.48
323,128	6.58	9.71	5.18	9.20	21.28	5.25	1071.66
490,351	6.60	9.72	5.11	9.13	22.58	6.07	1731.42
654,535	6.61	9.73	5.08	9.14	23.15	6.06	2483.67

Table 5.1: The results obtained on the Google Collection by the *B&B* algorithm. The results are expressed as bit per posting after the encoding phase. Interp. coding is the Binary Interpolative coding, while Var. Byte is the Variable Byte technique.

The above definition is very informal, since the optimality of an assignment also depends on the specific encoding method. However, we can observe that a practical simple measure of compressibility is the value of the *average gap* appearing in the various lists. When we reduce the average gap, the resulting IF results smaller almost independently from the encoding method actually adopted.

5.2.1 The *B&B* Algorithm

To complete the description of the background works of this chapter, we are going to explain and discuss some tests performed on the original implementation of the *B&B* algorithm².

Actually, this version is the one submitted to the past Google Programming Contest. Since, the tools and configuration files are all tailored over the Google Contest Document Collection (hereinafter Google Collection), all the results presented here will refer to this collection.

As stated above the *B&B* algorithm needs a sort of preprocessing step to build the *Reduced Assignment Graph G*. This required a running of a *Sort Based* inversion algorithm on the collection of documents. This initial step produces a file containing *G*.

The results of the tests conducted are in table 5.1. They represent the number of bits required to store each postings in the IF lists. Further, to evaluate the scalability of the method we varied also the collection size. We performed the tests on collections with a number of documents ranging from about 82,000 documents to about 654,000 distinct documents.

From these results several observations could be pointed out. The first regard the effectiveness of *B&B* in producing gains in compression-ratio of the Binary Interpolative Coding. In this case, *B&B* obtain enhancements up to 23.15% with respect to the case in which the docIDs have been assigned randomly. Another interesting property of the *B&B*

²The tests have been performed using the experimental setup described in section 5.5. We thank Blleloch and Blandford for kindly providing us the source code of their implementation.

algorithm is that the larger the collection the greater the gain in effectiveness. This fact was also pointed out in the paper by Blelloch [15] but the reordering time for a collection of about 565,000 documents is quite high. Note that the times reported in the table above do not consider the time spent in building the IF structure used by the algorithm itself.

We also empirically evaluated the memory required by the reordering process and we found that for the largest collection more than 1.6GByte of main memory were used. Nevertheless, the main drawback of the *B&B* approach is that it does not provide a method to assign the identifiers to the documents while they arrive to the indexer but it must rely on an already built IF index structure and then reorder its posting lists.

Further, the data model exploited by this algorithm requires the rebuilding of a new IF-like structure at each iteration.

We also tested the scalability of *B&B* with the number of documents to reorder. To do so we measured the completion time when varying the number of documents to rearrange. Figure 5.3 shows the results of these tests. As it can be seen, the plotted curve follows a $n \log(n)$ trend, where n is the number of documents. To assess our observation we then evaluated the theoretical upper bound of the complexity of the *B&B* algorithm.

We already pointed out the main steps of the algorithm that are:

1. Sampling the graph.
2. Bisecting the graph.
3. Assigning the unsampled documents to the best cluster found.
4. Reapplying the algorithm on the two found partition.
5. Choosing the best order of the two partition.

For our analysis purposes let's assume that after the first sampling step we have a scaled down *terms-documents* bipartite graph $\tilde{G} = (\tilde{V}, \tilde{E})$ with $|\tilde{V}| = N_{\rho,\tau}$, where $N_{\rho,\tau}$ is the number of documents remained after this initial step. Moreover let $T_{\text{Metis}}(N_{\rho,\tau})$ be the time spent by the *Metis* [85] algorithm to compute a two-way splitting of the graph \tilde{G} . By simply summing the cost of each of the *B&B* algorithm we obtain the following recurrence formula for its complexity:

$$\begin{aligned}
 T(N) &= T_{\text{Metis}}(N_{\rho,\tau}) + 2 \cdot N_{\rho,\tau} + 2 \cdot (N - N_{\rho,\tau}) + 2 \cdot T\left(\frac{N}{2}\right) = \\
 &= T_{\text{Metis}}(N_{\rho,\tau}) + 2 \cdot N_{\rho,\tau} + 2 \cdot N - 2 \cdot N_{\rho,\tau} + 2 \cdot T\left(\frac{N}{2}\right) = \\
 &= T_{\text{Metis}}(N_{\rho,\tau}) + 2 \cdot N + 2 \cdot T\left(\frac{N}{2}\right) > \\
 &> 2 \cdot N + 2 \cdot T\left(\frac{N}{2}\right) = \\
 &= N \cdot \log(N)
 \end{aligned}$$

for this reason the complexity of the *B&B* algorithm is $\Omega(N \cdot \log(N))$.

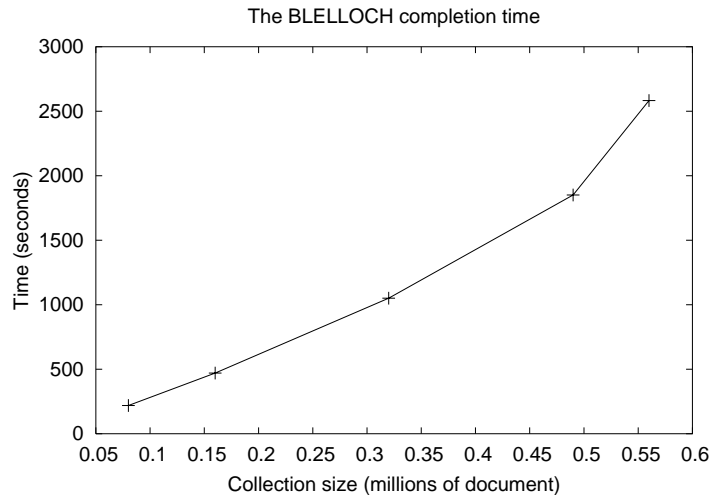


Figure 5.3: The scalability of the *B&B* algorithm when varying the size of the collection reordered.

Just for the sake of completeness we must say that the actual *B&B* algorithm complexity is higher and this is mainly due to two reasons. First *B&B* needs an IF to be built in order to start its computations. Then it needs to load the entire IF into main memory before starting the actual reordering phase.

To conclude, the *B&B* algorithm is very good for computing an optimal reordering of the document identifiers in order to reduce the space occupancy of a compressed IF but, to do so, it must first build a sort of Inverted List Index before starting. This step should be avoided in real WEB Search Engine Systems.

The observations made above may bring us to consider the problem from another point of view. In fact we would not start from an already built IF. Instead, we would like to *assign* identifiers to documents as they arrive to the indexers.

5.3 Collection Model

Differently from the approaches proposed so far, our algorithms adopt a collection model which does not assume the existence of a previously built IF index. All the algorithms take in input a *transactional* representation of the documents belonging to the collection. Each document $d_i \in \mathcal{D}$ is represented by the set $\tilde{d}_i \subseteq T$ of all the terms that the document contains. We denote the collection of documents in its transactional form with $\tilde{\mathcal{D}} = \{\tilde{d}_1, \tilde{d}_2, \dots, \tilde{d}_{|\mathcal{D}|}\}$. Basically, the transactional representation is very similar to the more popular bag-of-words one, but it does not consider the frequency of occurrences of the word within the documents (i.e. the *within-doc* frequency)

The transactional representation is actually stored by using a sort of digest scheme for each term. That is, for each \tilde{d}_i we store a list of integers obtained from the digest of the

terms contained within. In our case, for each term we proceed as follows: we compute the MD5 digest [1] and pick the first four bytes of the computed digest. Since the MD5 features a very low collision rate for arbitrary long texts, it is very likely that it remains true even considering just the first four bytes of the digest. To support our claim, we tested this digest scheme on the terms contained in the Google collection. We measured a collision ratio of 0.0006083, i.e. roughly a collision every thousand distinct terms.

Finally, we used the *Jaccard* measure [59] to evaluate document similarity. The Jaccard measure is a well known metric used in cluster analysis to estimate the similarity between two generic objects described by the presence or absence of attributes. It counts the number of attributes common to both objects, and divides this number by the number of attributes owned by at least one of them. More formally, the *Jaccard* measure used to measure the similarity between two distinct documents \tilde{d}_i , and \tilde{d}_j of $\tilde{\mathcal{D}}$ is given by:

$$jaccard_measure(\tilde{d}_i, \tilde{d}_j) = \frac{|\tilde{d}_i \cap \tilde{d}_j|}{|\tilde{d}_i \cup \tilde{d}_j|}$$

5.4 Our Algorithms

From a first analysis of the problem we could devise two different assignment schemes:

- *top-down assignment*: we start from the collection as a whole, and we recursively partition it by assigning, at each level, similar documents to the same partition. At the end of this partitioning phase a merging phase is performed until a single and ordered group of documents is obtained. The assignment function π is then deduced by the ordering of this last single group. This is the approach also followed by *B&B*. Within this scheme we propose two different algorithms which will be discussed in the following: TRANSACTIONAL *B&B* and *Bisecting*;
- *bottom-up assignment*: we start from a flat set of documents and extract from this set disjoint sequences containing similar documents. Inside each sequence the documents are ordered, while we do not make any assumption on the precedence relation among documents belonging to different sequences. The assignment function π in this case is deduced by first considering an arbitrary ordering of the produced sequences and then the internal ordering of the sequences themselves. In our case to order the produced sequences we simply consider the same order in which the sequences are produced by the algorithms themselves. Within this approach we propose two different algorithms: single-pass *k-means* and *k-scan*.

5.4.1 Top-down assignment

In the top-down scheme we start from the set $\tilde{\mathcal{D}}$. The general scheme of our top-down algorithms is the following (see Algorithm 1):

1. *center selection* (steps 3-5 of algorithm 1): according to some heuristic H , we select two (groups of) documents from \tilde{D} which will be used as partition representatives during the next step;
2. *redistribution* (steps 6-18) : according to their similarity to the centers, we assign each unselected document to one of the two partitions \tilde{D}' and \tilde{D}'' . Actually, we adopt a simple heuristic which consists in assigning exactly $\frac{|\tilde{D}|}{2}$ documents to each partition in order to equally split the computational workload among the two partitions;
3. *recursion* (steps 19-20): we recursively call the algorithm on the two resulting partitions until each partition becomes a singleton;
4. *merging* (steps 21-25): the two partitions built at each recursive call are merged (operator \oplus) bottom-up thus establishing an ordering (\preceq) between them. The precedence relation \preceq is obtained by comparing the borders of the partitions to merge (\tilde{D}' and \tilde{D}'') and, according to the distance measure adopted, we put \tilde{D}' before \tilde{D}'' if the similarity between the last document(s) of \tilde{D}' and the first document(s) of \tilde{D}'' is greater than the similarity computed by swapping the two partitions.

It is also possible to devise a general cost scheme for such top-down algorithms.

Claim 1. *Let \tilde{D} be a collection of documents. The cost of our top-down assignment algorithms is*

$$T(|\tilde{D}|) = O(|\tilde{D}| \log(|\tilde{D}|))$$

Proof. Let σ be the cost of computing the Jaccard distance between two documents, and τ the cost of comparing two Jaccard measures. Computing the Jaccard similarity mainly consists in performing the intersection of two sets, while comparing the similarity of two document only requires to compare two floats. We thus have that $\sigma \gg \tau$. Furthermore, let C_H be the cost of the heuristic H used to select the initial centers, and C_S be the cost of the *merging* step.

At each iteration, the top-down algorithm computes the initial centers. Then it computes at most $|\tilde{D}| - 2$ Jaccard distances in order to assign each document to the right partition. The total cost of this phase at each iteration is thus bounded by: $\sigma(|\tilde{D}| - 2) + C_H$.

At the end of the *center selection* and *distribution* phases, the top-down algorithm proceeds by calling recursively itself on the two equally sized sub-partitions obtained so far (*recursion* step) and then proceeds to order and merge the two partitions obtained (*merging* step).

The total cost of the algorithm is thus given by the following recursive equation:

$$T(|\tilde{D}|) = C_H + \sigma(|\tilde{D}| - 2) + 2T\left(\frac{|\tilde{D}|}{2}\right) + C_S \quad (5.1)$$

Algorithm 1 $TDAssign(\tilde{D}, H)$: the generic *top-down* assignment algorithm.

```

1: Input:
    • The set  $\tilde{D}$ .
    • The function  $H$  used to select the initial documents to form the centers of mass of the partitions.

2: Output:
    • An ordered list representing an assignment function  $\pi$  for  $\tilde{D}$ .

3:  $(\tilde{D}', \tilde{D}'') = H(\tilde{D})$ ;
4:  $c_1 = \text{center\_of\_mass}(\tilde{D}')$ ;
5:  $c_2 = \text{center\_of\_mass}(\tilde{D}'')$ ;
6: for all not previously selected  $d \in \tilde{D} \setminus (\tilde{D}' \cup \tilde{D}'')$  do
7:   if  $(|\tilde{D}'| \geq \frac{|\tilde{D}|}{2}) \vee (|\tilde{D}''| \geq \frac{|\tilde{D}|}{2})$  then
8:     Assign  $d$  to the smallest partition;
9:   else
10:     $dist_1 = \text{distance}(c_1, d)$ ;
11:     $dist_2 = \text{distance}(c_2, d)$ ;
12:    if  $dist_1 < dist_2$  then
13:       $\tilde{D}' = \tilde{D}' \cup \{d\}$ ;
14:    else
15:       $\tilde{D}'' = \tilde{D}'' \cup \{d\}$ ;
16:    end if
17:  end if
18: end for
19:  $\tilde{D}'_{ord} = TDAssign(\tilde{D}', H)$ ;
20:  $\tilde{D}''_{ord} = TDAssign(\tilde{D}'', H)$ ;
21: if  $\tilde{D}'_{ord} \preceq \tilde{D}''_{ord}$  then
22:    $\tilde{D}_{ord} = \tilde{D}'_{ord} \oplus \tilde{D}''_{ord}$ 
23: else
24:    $\tilde{D}_{ord} = \tilde{D}''_{ord} \oplus \tilde{D}'_{ord}$ 
25: end if
26: return  $\tilde{D}_{ord}$ ;

```

This equation corresponds to the well known:

$$T(|\tilde{D}|) = O(|\tilde{D}| \log(|\tilde{D}|))$$

□

Furthermore, we can compute the space occupied by the top-down assignment algorithm.

Claim 2. *The space occupied by our top-down assignment algorithm is given by*

$$S(|\tilde{D}|) = O(|\tilde{D}| \log(|\tilde{D}|))$$

Proof. Since we need to keep, at each level, a bit indicating the assigned partition, we need in total $S_{rec}(|\tilde{D}|) = |\tilde{D}| + S_{rec}\left(\frac{|\tilde{D}|}{2}\right) = O(|\tilde{D}| \log |\tilde{D}|)$ bits to store the partition assignment map. In practice $S_{rec}(|\tilde{D}|)$ defines the total space occupied by the partitions D' and D'' at all levels.

Now, let $|\bar{S}|$ be the average length of a document. The total space of the algorithm is thus given by:

$$S(|\tilde{D}|) = |\bar{S}||\tilde{D}| + \text{documents} \\ + 2S_{rec}(|\tilde{D}|) \text{ space for } D' \text{ and } D''$$

We can get rid of the linear term thus obtaining:

$$S(|\tilde{D}|) = O(|\tilde{D}| \log(|\tilde{D}|)) \quad (5.2)$$

□

As for the time, also the space complexity of the algorithm is super-linear in the number of documents processed. In practice, anyway, this is not a correct assertion. In fact the linear term dominates the $n \log n$ one until $4 \cdot \log n \leq 1000$. The last value for which the inequality holds is given by $\log n \leq 250 \Leftrightarrow n \leq 2^{250}$. Obviously the size of the whole Web is considerably smaller than 2^{250} documents!

We designed two different top-down algorithms: TRANSACTIONAL *B&B* and *Bisecting*.

TRANSACTIONAL *B&B*

The TRANSACTIONAL *B&B* algorithm is basically a porting under our model of the algorithm described in [15]. We briefly recall how the original *B&B* algorithm works. It starts by computing a sampled similarity graph: it chooses a document out of $|\tilde{D}|^\rho$ (ρ is the document sampling factor $0 < \rho < 1$) only considering terms appearing in less than τ documents. After this reduced similarity graph has been built, it applies the *Metis* graph partitioning algorithm [85], which splits the graph in two equally sized partitions. The algorithm then proceeds with the *redistribution*, *recursion*, and *merging* steps of the generic top-down algorithm. However, since in our model we do not have an IF index previously built over the document collection, we cannot know which terms appear in less than τ documents, and thus we did not introduce sampling over the maximum term frequency as in the original implementation.

In TRANSACTIONAL *B&B* the cost C_H at each iteration is thus given by the cost of picking up a subset of documents with a sampling factor equal to ρ , plus the cost of building the distance graph over this subset and computing the *Metis* algorithm over this graph.

Bisecting

The second algorithm we propose is called *Bisecting*. In this algorithm we adopt a *center selection* step which simply consists of uniformly choosing *two* random documents as centers. The cost of the *centers selection* step is thus reduced considerably. The algorithm is based on the simple observation that, since in TRANSACTIONAL *B&B* the cost C_H may be high, the only way to reduce it is to choose a low sampling parameter ρ , thus selecting at each iteration a very small number of documents as centers of the partitions. Thus we thought to just get rid of the first three phases, i.e. sampling, graph building, and *Metis* steps.

5.4.2 Bottom-up assignment

These algorithms consider each document of the collection separately, and proceed by progressively grouping together similar documents. Our bottom-up algorithms thus produce a set of non-overlapping sequences of documents.

The two different assignment algorithms presented here are both inspired by the popular *k-means* clustering algorithm [26]:

- a single-pass *k-means* algorithm;
- *k-scan* which is based on a centroid search algorithm which adapts itself to the characteristics of the processed collection.

Single-pass *k-means*

k-means [26], is a popular iterative clustering techniques which defines a *Centroid Voronoi Tessellation* of the input space. The *k-means* algorithm works as follows. It initially chooses k documents as cluster representatives, and assigns the remaining $|\tilde{D}| - k$ documents to one of these clusters according to a given similarity metric. New centroids for the k clusters are then recomputed, and all the documents are reassigned according to their similarity with the new k centroids. The algorithm iterates until the position of the k centroids become stable. The main strength of this algorithm is the $O(|\tilde{D}|)$ space occupancy. On the other hand, computing the new centroids is expensive for large values of $|\tilde{D}|$, and the number of iterations required to converge may be high. The single-pass *k-means* consists of just the first pass of this algorithm where the k centers are chosen using the technique described in [42]: *Buckshot*. We will not describe here the *Buckshot* technique, the only thing to keep into account is that the complexity of this step do not influence the theoretical linear performance of *k-means* which remains $O(k|\tilde{D}|)$. Since the *k-means* algorithm does not produce ordered sequences but just clusters, the internal order of each cluster is given by the insertion order of documents into each cluster.

***k*-scan**

The other bottom-up algorithm developed is *k*-scan. It resembles to the *k*-means one. It is, indeed, a simplified version requiring only *k* steps. At each step *i*, the algorithm selects a document among those not yet assigned and uses it as centroid for the *i*-th cluster. Then, it chooses among the unassigned documents the $\frac{|\tilde{D}|}{k} - 1$ ones most similar to the current centroid and assign them to the *i*-th cluster. The time and space complexity is the same as the single-pass *k*-means one and produces sets of ordered sequences of documents. Such ordering is exploited to assign consecutive DocIDs to consecutive documents belonging to the same sequence. The *k*-scan algorithm is outlined in Algorithm 2. It takes as input parameters the set \tilde{D} , and the number *k* of sequences to create. It outputs the ordered list of all the members of the *k* clusters. This list univocally defines π , an assignment of \tilde{D} minimizing the average value of the *d*-gaps.

Algorithm 2 The *k*-scan assignment algorithm.

```

1: Input:
   • The set  $\tilde{D}$ .
   • The number k of sequences to create.

2: Output:
   • k ordered sequences representing an assignment  $\pi$  of  $\tilde{D}$ .

3: sort  $\tilde{D}$  by descending lengths of its members;
4:  $c_i = \emptyset \quad i = 1, \dots, k$ ;
5: for  $i = 1, \dots, k$  do
6:    $current\_center = longest\_member(\tilde{D})$ 
7:    $\tilde{D} = \tilde{D} \setminus current\_center$ 
8:   for all  $\tilde{d}_j \in \tilde{D}$  do
9:      $sim[j] = compute\_jaccard(current\_center, \tilde{d}_j)$ 
10:  end for
11:   $M = select\_members(sim)$ 
12:   $c_i = c_i \oplus M$ 
13:   $\tilde{D} = \tilde{D} \setminus M$ 
14:   $c_i = c_i \oplus current\_center$ 
15:   $dump(c_i)$ 
16: end for
17: return  $\bigoplus_{i=1}^k c_i$ ;

```

The algorithm performs *k* scans of \tilde{D} . At each scan *i*, it chooses the longest document not yet assigned to a cluster as current center of cluster c_i , and computes the distances between it and each of the remaining unassigned documents. Once all the similarities have been computed, the algorithm selects the $\left(\frac{|\tilde{D}|}{k}\right) - 1$ documents most similar to the current center by means of the procedure reported in Algorithm 3, and put them in c_i . It is worth noting that when two documents result to have the same similarity, the longest one is selected. In fact, since the first DocID of each posting list has to be coded as it is, assigning smaller identifiers to documents containing a lot of distinct terms, maximizes

Algorithm 3 The *select_members* procedure.

```

1: Input:
    • An array sim: sim[j] contains the similarity between current_center and Sj.

2: Output:
    • The set of the  $\left(\left\lceil \frac{|\tilde{D}|}{k} \right\rceil - 1\right)$  documents more similar to current_center.

3: Initialize a min_heap of size  $\left(\left\lceil \frac{|\tilde{D}|}{k} \right\rceil - 1\right)$ 

4: for  $i = 1, \dots, |\tilde{D}|$  do
5:   if (sim[i] > sim[heap_root()]) OR
      ((sim[i] = sim[heap_root()]) AND (length(i) > length(heap_root())))
      then
6:     heap_insert(i)
7:   end if
8: end for
9: M =  $\emptyset$ 
10: for  $i = 1, \dots, \left(\left\lceil \frac{|\tilde{D}|}{k} \right\rceil - 1\right)$  do
11:   M = M  $\oplus$  heap_extract()
12: end for
13: return M

```

the number of posting lists starting with small DocIDs.

The complexity of *k*-scan in terms of number of distance computation and in space occupied is given by the following two claims.

Claim 3. *The complexity of the k-scan algorithm is:*

$$T(|\tilde{D}|, k) = O\left(\left\lceil \frac{|\tilde{D}|}{k} \right\rceil\right)$$

Proof. Since we are focusing on the number of distance computations, the initial ordering step (at point 3) of Algorithm 2 should not be considered when computing the complexity of the algorithm. Let σ be the cost of computing the Jaccard similarity between two documents, and τ the cost of comparing two Jaccard measures. Computing the Jaccard similarity mainly requires to intersect two sets, while comparing two similarity measures only requires to compare two floats. We thus have that $\sigma \gg \tau$.

At each iteration, *k*-scan computes $\left\lceil \frac{|\tilde{D}|}{k} \right\rceil - i$ Jaccard measures. The total cost of this phase at each iteration *i* is thus:

$$\sigma \left(\left\lceil \frac{|\tilde{D}|}{k} \right\rceil - i \right)$$

Once all the entries in the vector of similarities *sim* have been computed, *k*-scan calls the *select_members* procedure which performs $\left\lceil \frac{|\tilde{D}|}{k} \right\rceil - i$ insertions into a heap of size $\left\lceil \frac{|\tilde{D}|}{k} \right\rceil - 1$. Since an insertion is actually performed only if the element in the root of the heap is smaller than the element to be inserted, we should scale down the cost by a factor

$\tau' \ll 1$. The total time spent in *select_members* is thus:

$$\omega \left(|\tilde{D}| - i \frac{|\tilde{D}|}{k} \right) \cdot \log \left(\frac{|\tilde{D}|}{k} - 1 \right) \quad \omega = \tau \cdot \tau'.$$

The total time spent by the algorithm is thus given by:

$$T(|\tilde{D}|, k) = \sum_{i=0}^{k-1} T'$$

where

$$T' = \sigma \left(|\tilde{D}| - i \frac{|\tilde{D}|}{k} \right) + \left(\omega \left(|\tilde{D}| - i \frac{|\tilde{D}|}{k} \right) \log \left(\frac{|\tilde{D}|}{k} - 1 \right) \right).$$

Since $\sigma \gg \tau \gg \omega$ we have that:

$$T(|\tilde{D}|, k) \approx \sum_{i=0}^{k-1} \sigma \left(|\tilde{D}| - i \frac{|\tilde{D}|}{k} \right) = \sigma |\tilde{D}| \left(\frac{k+1}{2} \right) = O(|\tilde{D}|k)$$

□

Claim 4. *The space occupied by the k -scan algorithm is:*

$$S(|\tilde{D}|, k) = O(|\tilde{D}|)$$

Proof. Let $|\bar{S}|$ be the average length of a document. The k -scan algorithm thus uses $|\bar{S}||\tilde{D}|$ words for storing the documents, $8|\tilde{D}|$ for the array of similarities and $4\frac{|\tilde{D}|}{k}$ for the heap data structure used by Algorithm 3.

The total space is thus given by

$$\begin{aligned} S(|\tilde{D}|, k) &= |\bar{S}||\tilde{D}| + && \text{documents} \\ &+ 8|\tilde{D}| + && \text{array of similarities} \\ &+ 4\frac{|\tilde{D}|}{k} + && \text{the heap data structure} \\ &= O(|\tilde{D}|) \end{aligned}$$

□

5.5 Experimental Setup

To assess the performance of our algorithms we tested them on a real collection of Web documents, the publicly available Google Programming Contest collection.³ The main characteristics of this collection are:

³http://www.google.com/programming_contest

- it contains about 916,000 documents coming from real web sites;
- it is monolingual;
- the number of distinct terms is about 1,435,000.

On the considered collection we performed a preprocessing step consisting in the transformation of the documents considered in the *transactional* model.

For each method proposed we evaluated: the completion time, the space occupied, and the compression ratios achieved after the assignment. The effectiveness gains resulting by adopting Binary Interpolative [105], Gamma [151] and Variable Byte [126] encoding methods were evaluated. We ran our tests on a Xeon 2GHz PC equipped with 1GB of main memory, and an Ultra-ATA 60GB disk. The operating system was Linux.

5.6 Comparisons

In Figure 5.4 the performance in terms of completion time (a), and space consumed (b) are shown. All the time reported are the actual times taken by all the algorithms to finish their operations and do not include I/O.

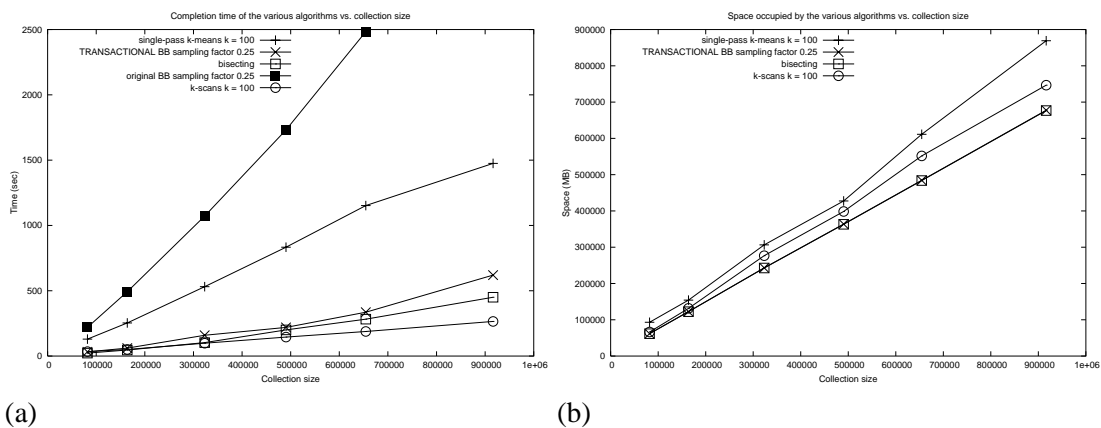


Figure 5.4: (a) Time (in seconds) and (b) Space (in KBytes) consumed by the proposed transactional assignment algorithms as a function of the collection size.

From Figure 5.4.(a) we can draw several important conclusions. First of all, the execution time of the original *B&B* algorithm is remarkably higher than the time spent by all our algorithms. In particular, on the whole Google contest collection (i.e. about 916,000 documents) the *B&B* algorithm ran out of memory before finishing its operations. Please note that the values plotted in the curve of the original *B&B* do not consider the time spent in preliminarily computing the input IF index. If we look at the curve related to the original implementation of the *B&B* algorithm we can observe an $n \log n$ behavior that is typical of top-down approaches described above. Looking at the curves of our algorithms,

it is evident that the single-pass k -means algorithm is the one that takes the highest time to compute the assignment. On the other hand, the others transactional techniques have relatively low completion times. In particular the k -scan algorithm sensibly outperforms the others. Obviously, the linear behavior exhibited by this algorithm evidentiates that on large collections k -scan will remarkably outperforms the others which instead show a $n \log n$ trend.

Figure 5.4.(b) shows the space occupancy of our algorithms. Since the tests with the original $B\&B$ algorithm were performed by using the implementation kindly provided to us by the authors, we were not able to measure directly the space occupied by this algorithm. However the algorithm is memory consuming and, as said above, on the whole Google contest collection it ran out of memory. Obviously, also in our case we cannot fit completely in memory large collections. In those cases we can split the collections in several partitions and then proceed to reorder each partition separately. The curves plotted in Figure 5.4.(b) show that, as expected, TRANSACTIONAL $B\&B$ and *Bisecting* use the same amount of memory and exhibit a linear scale-up trend. This last fact follows directly from the observations made in Section 5.4 about the space complexity of the top-down approaches. Anyway, the space occupied by all our algorithms appears to grow linearly with respect to the collection size.

The DocIDs of four collections of different size, with up to 916,000 documents, were assigned by exploiting the various algorithms. For each assignment we measured the compression performance. Table 5.2 reports the average number of bits required to represent each posting of the IF obtained after the DocID assignment with three popular encoding methods: *Interpolative*, *Gamma*, and *Variable Byte*. In all the cases we can observe a reduction in the average number of bits used with respect to a random DocID assignment, i.e., the baseline for comparisons reported in the first block of rows of the table. We can see that the original implementation of the $B\&B$ algorithm outperforms our algorithms. The gain in the compression performance of $B\&B$ is, in almost all the cases, about 10% which corresponds to ~ 0.5 bits saved for each posting. However, our methods spend remarkably less time than the $B\&B$ one. We can also observe that our methods are similar in terms of compression gain. For the largest collection the performance obtained is approximately the same for all the encoding methods and for all the assignment algorithms implemented. Moreover, we can note that the TRANSACTIONAL $B\&B$ algorithm obtains worse results than the original $B\&B$ algorithm. We think that this may depend on the term sampling which cannot be exploited by our TRANSACTIONAL $B\&B$. (see Section 5.4). The higher gains in compression performance were obtained by using the Gamma algorithm. This is a very important result since a method which is very similar to Gamma was recently presented in [6]. This method is characterized by a very good compression performance and a relatively low decoding overhead, in some cases lower than those of the Variable Byte method. The results on the Gamma algorithm are thus very important to validate our approaches.

Assignment Algorithm	Collection Size	Bits per Postings		
		Interpolative	Gamma	Var Byte
Random assignment	81,875	6.52	8.96	9.67
	323,128	6.59	9.05	9.71
	654,535	6.61	9.08	9.73
	916,429	6.61	9.10	9.72
<i>B&B</i>	81,875	5.31	6.71	9.17
	323,128	5.18	6.58	9.20
	654,535	5.08	6.40	9.14
	916,429	N/A	N/A	N/A
TRANSACTIONAL <i>B&B</i>	81,875	5.56	7.20	9.29
	323,128	5.46	7.11	9.32
	654,535	5.54	7.19	9.35
	916,429	6.04	8.04	9.52
<i>Bisecting</i>	81,875	5.66	7.60	9.37
	323,128	5.62	7.53	9.33
	654,535	5.71	7.66	9.38
	916,429	6.10	8.23	9.52
single-pass <i>k</i> -means	81,875	5.60	7.27	9.26
	323,128	5.64	7.34	9.32
	654,535	5.67	7.44	9.32
	916,429	6.10	8.11	9.51
<i>k</i> -scan	81,875	5.53	7.25	9.20
	323,128	5.56	7.36	9.27
	654,535	5.66	7.54	9.33
	916,429	6.10	8.11	9.51

Table 5.2: Performance, as average number of bits used to represent each posting, as a function of the assignment algorithm used, of the collection size (no. of documents), and of the encoding algorithm adopted. The row labeled “Random assignment” reports the performance of the various encoding algorithms when DocIDs are randomly assigned.

5.7 Summary

In this chapter we presented an analysis of several efficient algorithms for computing approximations of the optimal DocID assignment for a collection of textual documents. We have proved that our algorithms are a viable way to enhance the compressibility (up to 21%) of IF indexes.

The algorithms proposed operate following two opposite strategies: a top-down approach and a bottom-up approach. The first group includes the algorithms that recursively split the collection in a way that minimizes the distance of lexicographically closed documents. The second group contains algorithms which compute an effective reordering employing linear space and time complexities. Although our algorithms obtain gains in compression ratios which are slightly worse than those obtained by the *B&B* algorithm, their performance in terms of space and time are instead remarkably higher. Moreover, an appealing feature of our approach is the possibility of performing the assignment step on the fly, during the indexing process. As future work we plan to test the performance of our algorithms on some recently proposed encoding methods. In particular we would like to evaluate the method described in [6] for which we should be able to obtain good results. Furthermore, we want to investigate possible adaptations of the algorithms proposed to collections which change dynamically in the time.

Conclusions

The design and implementation, as well the analysis, of efficient, and effective Web Search Engines (WSEs), are becoming more and more important as the size of the Web has continually kept growing. Furthermore, the development of systems for Web Information Retrieval represents a very challenging task whose complexity imposes the knowledge of several concepts coming from many different areas: databases, parallel computing, artificial intelligence, statistics, etc.

In this thesis three important issues related to WSEs' technology have been investigated.

We presented a novel caching policy for the results produced by a WSE. We called this novel caching schema SDC (i.e. Static and Dynamic Caching). The schema proposed considers both the recency, and the frequency of the occurrences of queries arriving at a WSE. We experimentally compared SDC against all the most popular policies. We used, for our tests, three different query logs coming from three real world WSE: *Excite*, *Tiscali*, *Altavista*. In all the cases, SDC outperforms the other policies, at the same cost (or even lower) of a *LRU/SLRU* scheme. This last characteristics, indeed, is very important since many other policies enhance the *LRU/SLRU* schemes but, usually, need a logarithmic time complexity with respect to the size of the cache managed. Furthermore, SDC can be effectively adopted in a concurrent environment where, to enhance the throughput of the system, multiple clients access the cache at the same time. In this kind of scenario SDC fits very well since the accesses in the *Static Set* does not require mutex semaphores.

We presented indexing with the proposal of a novel software architecture exploiting the parallelism among the phases of the indexing process. We have discussed the design of WINGS, a parallel indexer for Web contents that produces local inverted indexes, the most commonly adopted organization for the index of a large-scale parallel/distributed WSE. WINGS exploits two different levels of parallelism. Data parallelism, due to the possibility of independently building separate inverted indexes from disjoint document partitions. This is possible because WSEs can efficiently process queries by broadcasting them to several searchers, each associated with a distinct local index, and by merging the results. In addition, we have also shown how a limited pipeline parallelism can be exploited within each instance of the indexer, and that a low-cost 2-way workstation equipped with an inexpensive IDE disk is able to achieve a throughput of about $2.7 \text{ GB}/\text{hour}$, when processing four document collections to produce distinct local inverted indexes.

We presented an analysis of several efficient algorithms for computing approximations

of optimal assignment for a collection of textual documents that effectively enhances the compressibility of the IF index built over the reordered collection. The algorithms shown operate following two opposite strategies: a top-down approach and a clustering approach. In the first group fall the algorithms that recursively split the collection in a way that minimizes the distance of lexicographically closed documents. The second group contains algorithms which compute an effective reordering employing linear space and time complexities. The experimental evaluation conducted with a real world test collection, resulted in improvements up to $\sim 23\%$ in the compression rate achieved.

Bibliography

- [1] RFC 1321. The md5 algorithm.
- [2] L. Adamic, R. Lukose, A. Puniyani, and B. Huberman. Search in Power-Law Networks. Available at <http://www.parc.xerox.com/istl/groups/iea/papers/plsearch/>, 2001.
- [3] Charu C. Aggarwal, Fatima Al-Garawi, and Philip S. Yu. Intelligent Crawling on the World Wide Web with Arbitrary Predicates. In *Proceedings of the World Wide Web 2001 (WWW10)*, pages 96–105, 2001.
- [4] Virgílio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the IEEE Conference on Parallel and Distributed Information Systems (PDIS)*, Miami Beach, FL, 1996.
- [5] Brian Amento, Loren G. Terveen, and Willuam C. Hill. Does “Authority” mean quality? Predicting expert quality ratings of web documents. *Research and Development in Information Retrieval*, pages 296–303, 2000.
- [6] V. N. Anh and A. Moffat. Index compression using fixed binary codewords. In K.-D. Schewe and H. Williams, editors, *Proc. 15th Australasian Database Conference*, Dunedin, New Zealand, January 2004.
- [7] V.N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 2004. To appear.
- [8] V.N. Anh and A. Moffat. Random Access Compressed Inverted Lists. In C. McDonald, editor, *Proceedings of the 9th Australasian Database Conference*, pages 1–12, Perth, February 1998. Springer-Verlag.
- [9] Yossi Azar, Amos Fiat, Anna R. Karlin, Frank McSherry, and Jared Saia. Spectral Analysis of Data. In *Proceedings of STOC 2001*, 2001.
- [10] C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In *Proc. of the 9th String Processing and Information Retrieval Symposium (SPIRE)*, September 2002.

- [11] Ricardo Baeza–Yates and Berthier Ribiero–Neto. *Modern Information Retrieval*. Addison Wesley, 1998.
- [12] Beowulf Project at CESDIS. <http://www.beowulf.org>.
- [13] M.W. Berry and M. Browne. *Understanding Search Engines: Mathematical Modeling and Text Retrieval*. SIAM Book Series: Software, Environments, and Tools, June 1999.
- [14] Krishna Bharat and Monika Rauch Henzinger. Improved Algorithms for Topic Distillation in a Hyperlinked Environment. *Research and Development in Information Retrieval*, pages 104–111, 1998.
- [15] Dan Blandford and Guy Blelloch. Index compression through document reordering. In IEEE, editor, *Proceedings of the DATA COMPRESSION CONFERENCE (DCC'02)*. IEEE, 2002.
- [16] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Trovatore: Towards a Highly Scalable Distributed Web Crawler. In *WWW Posters 2001*, 2001. <http://www10.org/cdrom/posters/1033.pdf>.
- [17] Allan Borodin, Gareth O. Roberts, Jeffrey S. Rosenthal, and Panayiotis Tsaparas. Finding Authorities and Hubs from Link Structures on the World Wide Web.
- [18] O. Brandman, J. Cho, H. Garcia-Molina, and N. Shivakumar. Crawler-Friendly Web Servers, 2000.
- [19] S. Brin and L. Page. The Anatomy of a Large–Scale Hypertextual Web Search Engine. In *Proceedings of the WWW7 conference / Computer Networks*, volume 1–7, pages 107–117, April 1998.
- [20] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.
- [21] Chris Buckley. Implementation of the smart information retrieval system. Technical Report TR85–686, Cornell University, Computer Science Department, May 1985.
- [22] Rajkumar Buyya, editor. *High Performance Cluster Computing*. Prentice Hall PTR, 1999.
- [23] Jamie Callan and Margaret Connell. Query–Based Sampling of Text Databases.
- [24] J.P. Callan, Z. Lu, and W.B. Croft. Searching Distributed Collections with Inference Networks. In E. A. Fox, P. Ingwersen, and R. Fidel, editors, *Proceedings of the Eighteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 21–28, Seattle, WA, July 1995. ACM Press.

- [25] S. Chakrabarti, B. Dom, D. Gibson, J. Kleimberg, S. R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Hypersearching the Web. *Scientific American*, June 1999.
- [26] Soumen Chakrabarti. *Mining the Web - Discovering Knowledge from Hypertext Data*. Morgan Kaufmann Publishers, San Francisco, 2003.
- [27] Soumen Chakrabarti, Byron Dom, Prabhakar Raghavan, Sridhar Rajagopalan, David Gibson, and Jon Kleimberg. Automatic Resource Compilation by Analyzing Hyperlink Structure and Associated Text. *Computer Networks and ISDN Systems*, 30(1–7):65–74, 1998.
- [28] Soumen Chakrabarti, Byron E. Dom, S. Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, Andrew Tomkins, David Gibson, and Jon Kleinberg. Mining the Web’s Link Structure. *Computer*, 32(8):60–67, 1999.
- [29] Soumen Chakrabarti, Martin van den Berg, and Byron Dom. Focused Crawling: A New Approach to Topic-Specific Web Resource Discovery. In *Proceedings of the 8th World Wide Web Conference*, Toronto, May 1999.
- [30] B. Chidlovskii, C. Roncancio, and M.L. Schneider. Semantic cache mechanism for heterogeneous web querying. In *Proc. of the 8th Int. World Wide Web Conf.*, 1999.
- [31] Boris Chidlovskii, Claudia Roncancio, and Marie-Luise Schneider. Semantic Cache Mechanism for Heterogeneous Web Querying. In *Proceedings of the WWW8 Conference / Searching and Querying*, 1999.
- [32] J. Cho and H. Garcia-Molina. Estimating Frequency of Change. Technical report, Stanford University, 2000.
- [33] Junghoo Cho. *Crawling the Web: Discovery and Maintenance of Large-Scale Web Data*. PhD thesis, Stanford University, October 2001.
- [34] Junghoo Cho and Hector Garcia-Molina. Synchronizing a Database to Improve Freshness. pages 117–128, 2000.
- [35] Junghoo Cho and Hector Garcia-Molina. The Evolution of the Web and Implications for an Incremental Crawler. In *The VLDB Journal*, pages 200–209, 2000.
- [36] Junghoo Cho, Hector García-Molina, and Lawrence Page. Efficient Crawling Through URL Ordering. *Computer Networks and ISDN Systems*, 30(71–7):161–172, 1998.
- [37] C. Clarke, G. Cormack, and F. Burkowski. Fast Inverted Indexes with On-Line Update. Technical report, University of Waterloo Computer Science Department, 1994. Available at <http://citeseer.nj.nec.com/clarke94fast.html>.

- [38] David Cohn. The Missing Link – a Probabilistic Model of Document Content and Hypertext Connectivity.
- [39] Keywords Distributed Collections. The Effects of Query-Based Sampling on Automatic Database Selection Algorithms.
- [40] N. Craswell, P. Bailey, and D. Hawking. Server Selection on the World Wide Web. In *Proceedings of the Fifth ACM Conference on Digital Libraries*, pages 37–46, 2000.
- [41] A. Crespo and H. Garcia-Molina. Routing Indices for Peer-to-Peer Systems. Submitted to ICDCS 2002, October 2001.
- [42] Douglass R. Cutting, David R. Karger, Jan O. Pedersen, and John W. Tukey. Scatter/gather: a cluster-based approach to browsing large document collections. In *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 318–329. ACM Press, 1992.
- [43] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.
- [44] S. Dar, M.J. Franklin, B. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proc. of the 22nd Int. Conf. on Very Large Database*, pages 330–341, 1996.
- [45] Jeffrey Dean and Monika Rauch Henzinger. Finding Related Pages in the World Wide Web. *WWW8 / Computer Networks*, 31(11–16):1467–1479, 1999.
- [46] Michelangelo Diligenti, Frans Coetzee, Steve Lawrence, C. Lee Giles, and Marco Gori. Focused Crawling using Context Graphs. In *Proceedings of the 26th International Conference on Very Large Databases, VLDB 2000*, pages 527–534, Cairo, Egypt, 10–14 September 2000.
- [47] R. Dolin, D. Agrawal, L. Dillon, and A. El Abbadi. Pharos: A Scalable Distributed Architecture for Locating Heterogeneous Information Sources. Technical Report TRCS96–05, 5 1996.
- [48] Petros Drineas, Alan Frieze, Ravi Kannan, Santosh Vempala, and V. Vinay. Clustering in Large Graph and Matrices. In *Proceedings of SODA: ACM–SIAM Symposium on Discrete Algorithms*.
- [49] J. Edwards, K. McCurley, and J. Tomlin. An Adaptive Model for Optimizing Performance of an Incremental Web Crawler. In *Proceedings of the World Wide Web 10 Conference WWW10*, pages 106–113, Hong Kong, 2001. <http://www10.org/cdrom/papers/pdf/p210.pdf>.

- [50] P. Elias. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, February 1975.
- [51] Christoph Baumgarten Email. A Probabilistic Solution to the Selection and Fusion Problem in Distributed Information Retrieval.
- [52] Excite Search Engine. <http://www.excite.com>.
- [53] Google Search Engine. <http://www.google.com>.
- [54] TodoBR Brazilian Search Engine. <http://www.todobr.com.br>.
- [55] Tiziano Fagni, Salvatore Orlando, Paolo Palmerini, Raffaele Perego, and Fabrizio Silvestri. A hybrid strategy for caching web search engine results. In *Proceedings of the twelfth international conference on World Wide Web*, 2003.
- [56] T. Feder, R. Motwani, R. Panigrahy, S. Seiden, R. van Stee, and A. Zhu. Web caching with request reordering. In *Proceedings of the 13th Annual Symposium on Discrete Algorithms*, pages 104–105, 2002.
- [57] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan-Kaufmann, 1999.
- [58] W. Frakes and R. Baeza-Yates. *Information Retrieval*. Prentice–Hall, Englewood Cliffs, NJ, 1992.
- [59] W. B. Frakes and Editors R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*, chapter Clustering Algorithms (E. Rasmussen). Prentice Hall, Englewood Cliffs, NJ, 1992.
- [60] James C. French, Allison L. Powell, James P. Callan, Charles L. Viles, Travis Emmitt, Kevin J. Prey, and Yun Mou. Comparing the Performance of Database Selection Algorithms. In *Research and Development in Information Retrieval*, pages 238–245, 1999.
- [61] James C. French, Allison L. Powell, and Jamie Callan. Effective and Efficient Automatic Database Selection. Technical Report CS-99-08, 2 1999.
- [62] James C. French, Allison L. Powell, Charles L. Viles, Travis Emmitt, and Kevin J. Prey. Evaluating Database Selection Techniques: A Testbed and Experiment. In *Research and Development in Information Retrieval*, pages 121–129, 1998.
- [63] N. Fuhr. A Decision–Theoretic Approach to Database Selection in Networked IR. In Workshop on Distributed IR, 1996.
- [64] N. Fuhr. Optimum Database Selection in Networked IR. In *Proceedings of the SIGIR’96 Workshop Networked Information Retrieval*, Zurich, Switzerland, 1996.

- [65] S. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 12(3):399–401, July 1966.
- [66] L. Gravano and H. Garcia–Molina. Generalizing GLOSS to Vector–Space Databases and Broker Hierarchies. In *Proceedings of the 21st VLDB Conference*, Zurich, Switzerland, 1995.
- [67] L. Gravano, H. Garcia–Molina, and A. Tomasic. Precision and Recall of GLOSS Estimators for Database Discovery. Stanford University Technical Note Number STAN-CS-TN-94-10, 1994.
- [68] L. Gravano, H. Garcia–Molina, and A. Tomasic. The Effectiveness of GLOSS for the Text Database Discovery Problem. In *Proceedings of the SIGMOD 94*. ACM, September 1994.
- [69] Luis Gravano, Hector Garcia-Molina, and Anthony Tomasic. The Efficacy of GLOSS for the Text Database Discovery Problem. Technical Report CS-TN-93-2.
- [70] Michael Gorlick Gregory Alan Bolcer, Arthur S. Hitomi, Peter Kammer, Brian Morrow, Peyman Oreizy, and Richard N. Taylor. *Peer-to-Peer Architectures and the Magi Open-Source Infrastructure*. Endeavors Technology, Inc., Irvine, California 92612, December 2000.
- [71] Vijay Gupta and Roy Campbell. Internet Search Engine Freshness by Web Server Help. In *Proceedings of the 2001 Symposium on Applications and the Internet (SAINT 2001)*. IEEE, 2001.
- [72] David Hawking and Paul Thistlewaite. Methods for Information Server Selection. *ACM Transactions on Information Systems*, 17(1):40–76, 1999.
- [73] Allan Heydon and Marc Najork. Mercator: A Scalable, Extensible Web Crawler. *World Wide Web*, 2(4):219–229, 1999.
- [74] C. Hoelscher. How internet experts search for information on the web. Paper presented at the World Conference of the World Wide Web, Internet, and Intranet, Orlando, FL, 1998.
- [75] Thomas Hofmann. Probabilistic Latent Semantic Indexing. *Research and Development in Information Retrieval*, 1999.
- [76] S. Tuecke I. Foster, C. Kesselman. The Anatomy of the Grid: Enabling Scalable Virtual Organization. *Int’l Journal on Supercomputer Application*, 3(15).
- [77] Paul Ogilvie Jamie. The Effectiveness of Query Expansion for Distributed Information Retrieval.

- [78] Bernard J. Jansen and Udo W. Pooch. A review of web searching studies and a framework for future research. *Journal of the American Society of Information Science*, 52(3):235–246, 2001.
- [79] Bernard J. Jansen, Amanda Spink, and Tefko 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.
- [80] B.J. Jansen, A. Spink, J. Bateman, and T. Saracevic. Searchers, the subjects they search and sufficiency: a study of a large sample of EXCITE searches. In *Proceedings of WebNet 98*, 1998.
- [81] B.S. Jeong and E. Omiecinski. Inverted File Partitioning Schemes in Multiple Disk Systems. *IEEE Transactions on Parallel and Distributed Systems*, (2), February 1995.
- [82] Trevor Jim and Dan Suciu. Dynamically Distributed Query Evaluation. In *Proceedings of the PODS 2001 conference*, Santa Barbara, CA, May 2001. ACM.
- [83] Theodore Johnson and Dennis Shasha. 2q: A low overhead high performance buffer management replacement algorithm. In *Proc. 1994 Very Large Data Bases*, pages 439–450, 1994.
- [84] Project JXTA. <http://www.jxta.org>.
- [85] George Karypis. Metis: Family of multilevel partitioning algorithms. <http://www-users.cs.umn.edu/karypis/metis/>.
- [86] Jon M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *Journal of the ACM*, 46(5):604–632, 1999.
- [87] Donald Kossmann. The State of the Art in Distributed Query Processing. *ACM Computer Survey*, 1999.
- [88] R. Krikorian. Hello jxta! available at <http://www.onjava.com/lpt/a/onjava/2001/04/25/jxta.html>, 2001.
- [89] J. Kubiawicz, D. Bindel, Y. Chen, S. Czewinsky, P. Eaton, D. Geels, R. Gum-madi, S. Thea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: an Architecture for Global-Scale Persisten Storage. In *Proceedings ASPLOS*, pages 190–201, November 2001.
- [90] Leah S. Larkey, Connel, Margaret, and Jamie Callan. Collection selection and results merging with topically organized u.s. patents and trec data. To appear in *Proceedings of Ninth International Conference on Information Knowledge and Man-agement*, November 6–10 2000.

- [91] Steve Lawrence and C. Lee Giles. Searching the World Wide Web. *Science*, 280(5360):98–100, 1998.
- [92] R. Lempel and S. Moran. SALSA: The Stochastic Approach for Link-Structure Analysis. *ACM Transactions on Information Systems*, 19(2):131–160, 2001.
- [93] Ronny Lempel and Shlomo Moran. Optimizing results prefetching in web search engines with segmented indices. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, 2002.
- [94] Ronny Lempel and Shlomo Moran. Predictive caching and prefetching of query results in search engines. In *Proceedings of the twelfth international conference on World Wide Web*, pages 19–28. ACM Press, 2003.
- [95] Z. Lu, J. Callan, and W. Croft. Measures in Collection Ranking Evaluation, 1996.
- [96] Zhihong Lu and Kathryn S. McKinley. Partial replica selection based on relevance for information retrieval. In *SIGIR '99: Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 15-19, 1999, Berkeley, CA, USA*, pages 97–104. ACM, 1999.
- [97] Zhihong Lu and K.S. McKinley. *Advances in Information Retrieval*, chapter 7. The Effect of Collection Organization and Query Locality on Information Retrieval System Performance and Design. Kluwer, New York. Bruce Croft Editor, 2000.
- [98] G. Manku and R. Motwani. Approximate frequency counts over data streams, 2002.
- [99] Evangelos P. Markatos. On caching search engine results. In *Proc. of the 5th Int. Web Caching and Content Delivery Workshop*, 2000.
- [100] Evangelos P. Markatos. On Caching Search Engine Query Results. *Computer Communications*, 24(2):137–143, 2001.
- [101] Andrew McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Building Domain-Specific Search Engines with Machine Learning Techniques. In *Proceedings of the AAAI Spring Symposium on Intelligent Agents in Cyberspace 1999*, 1999.
- [102] Sergey Melnik, Sriram Raghavan, Beverly Yang, and Hector Garcia-Molina. Building a Distributed Full-Text Index for the Web. In *World Wide Web*, pages 396–406, 2001.
- [103] A. Moffat and T. Bell. In situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537–550, 1995.

- [104] A. Moffat and J. Zobel. Self-Indexing Inverted Files for Fast Text Retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, October 1999.
- [105] Alistair Moffat and Lang Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, July 2000.
- [106] Alistair Moffat and Justin Zobel. Information Retrieval Systems for Large Document Collections. In *Proceedings of the Text REtrieval Conference*, pages 85–94, 1994.
- [107] S. Mukherjea. WTMS: A System for Collecting and Analyzing Topic-Specific Web Information. *Computer Networks*, 33(1):457–471, 2000.
- [108] M. Najork and A. Heydon. On High-Performance Web Crawling.
- [109] Marc Najork and Janet L. Wiener. Breadth-First Crawling Yields High-Quality Pages. In *Proceedings of the WWW 2001 conference*, pages 114–118, 2001.
- [110] Anh NgocVo and Alistair Moffat. Compressed inverted files with reduced decoding overheads. In *Proc. of the 21st Intl. Conf. on Research and Development in Information Retrieval*, pages 290–297, October 1998.
- [111] Elisabeth J. O’Neil, Patrick E. O’Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffer. In *Proceedings of the 1993 ACM Sigmod International Conference On Management Of Data*, pages 297–306, 1993.
- [112] S. Orlando, R. Perego, and F. Silvestri. Design of a parallel and distributed web search engine. In *The 2001 Parallel Computing Conference (ParCo 2001)*, 2001.
- [113] Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. Design of a Parallel and Distributed WEB Search Engine. In *Proceedings of Parallel Computing (ParCo) 2001 conference*. Imperial College Press, September 2001.
- [114] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The Pagerank Citation Ranking: Bringing Order to the Web. Technical report, Stanford University, 1998.
- [115] A. Powell, J. French, J. Callan, Connell, and C. M. Viles. The Impact of Database Selection on Distributed Searching. In *Proceedings of the SIGIR 2000 conference*, pages 232–239. ACM, 2000.
- [116] Sriram Raghavan and Hector Garcia-Molina. Crawling the hidden web. In *The VLDB Journal*, pages 129–138, 2001.
- [117] Anand Rajaraman and Jeffrey D. Ullman. Querying Websites Using Compact Skeletons. In *Proceedings of the PODS 2001 conference*, Santa Barbara, CA, May 2001. ACM.

- [118] Yves Rasolofo. Approaches to Collection Selection and Results Merging for Distributed Information Retrieval.
- [119] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings ACM SIGCOMM*, August 2001.
- [120] B. Ribeiro-Neto, E. S. Moura, M. S. Neubert, and N. Ziviani. Efficient Distributed Algorithms to Build Inverted Files. In *Proceedings of 22nd ACM Conference on R&D in Information Retrieval*, August 1999.
- [121] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *Proc. of the 1990 ACM SIGMETRICS Conference*, pages 134–142, 1990.
- [122] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.
- [123] A. Rowstron and P. Druschel. Storage Management and Caching in Past, Large-Scale, Persistent Peer-to-Peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.
- [124] Patricia Correia Saraiva, Edleno Silva de Moura, Nivio Ziviani, Wagner Meira, Rodrigo Fonseca, and Berthier Ribeiro-Neto. Rank-Preserving Two-Level Caching for Scalable Search Engines. In ACM, editor, *Proceedings of the SIGIR2001 conference*, New Orleans, LA, September 2001. SIGIR.
- [125] P.C. Saraiva, E. Silva de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. Ribeiro-Neto. Rank-preserving two-level caching for scalable search engine. In *SIGIR'01*, 2001.
- [126] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted index for fast query evaluation. In *SIGIR02*, 2002.
- [127] Yipeng Shen, Dik Lun Lee, and Lian Wen Zhang. A Distributed Search System Based on Markov Decision Processes. In *Proceedings of the ICSC 99 conference*, Honk Kong, December 1999.
- [128] Wann-Yun Shieh, Tien-Fu Chen, Jean Jyh-Jiun Shann, and Chung-Ping Chung. Inverted file compression through document identifier reassignment. *Information Processing and Management*, 39(1):117–131, January 2003.
- [129] Bilal Siddiqui. Using UDDI as a Search Engine – Smart Web Crawlers For All. <http://www.webservicesarchitect.com/content/articles/siddiqui01.asp>, October 2001.

- [130] C. Silverstein, M. Henzinger, H. Marais, and M. Moricz. Analysis of a Very Large AltaVista Query Log. Technical report, Digital Inc., October 1998. SRC Technical Note 1998–14.
- [131] C. Silverstein, M. Henzinger, H. Marais, and M. Moricz. Analysis of a very large web search engine query log. In *ACM SIGIR Forum*, pages 6–12, 1999.
- [132] Fabrizio Silvestri, Raffaele Perego, and Salvatore Orlando. Assigning document identifiers to enhance compressibility of web search engine indexes. In *Proceedings of the 19th Annual ACM Symposium on Applied Computing - Data Mining Track*, March 2004.
- [133] Fabrizio Silvestri, Raffaele Perego, and Salvatore Orlando. Assigning identifiers to documents to enhance the clustering property of fulltext indexes. In *Proceedings of the 27th Annual International ACM SIGIR Conference*, July 2004.
- [134] Gnutella Web site. <http://gnutella.wega.com>.
- [135] Morpheus Web site. <http://www.musiccity.com>.
- [136] Napster Web site. <http://www.napster.com>.
- [137] NeuroGrid Web site. <http://www.neurogrid.net>.
- [138] XML Web site. <http://www.w3.org/tr/rec-xml>.
- [139] T. B. Smith, B. Abali, D. E. Poff, and R. B. Tremaine. Memory expansion technology (mxt): Competitive impact. *Journal of Research and Development*, 45(2):303–309, February 2001.
- [140] A. Spink, B.J. Jansen, and J. Bateman. Users’ searching behavior on the EXCITE web search engine. In *Proceedings of WebNet 98*, 1998.
- [141] A. Spink, B.J. Jansen, D. Wolfram, and T. Saracevic. Searching the web: the public and their queries. *J. Am. Soc. Information Science and Technology*, 53(2):226–234, 2001.
- [142] A. Spink, B.J. Jansen, D. Wolfram, and T. Saracevic. From e-sex to e-commerce: Web search changes. *Computer*, 35(3):107–109, March 2002.
- [143] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: a Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM*, August 2001.
- [144] P. Thati, P. H. Chang, and G. Agha. Crawlets: Agents for High Performance Web Search Engines. In *Proceedings of the 5th IEEE International Conference on Mobile Agents (MA 2001)*. IEEE, December 2001.

- [145] Doug Tidwell. Web services – the Web’s next revolution. <http://www-105.ibm.com/developerworks/education.nsf/webservices-onlinecourse-bytitle/BA84142372686CFB862569A400601C18?OpenDocument>, November 2000. (Registration required).
- [146] A. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental Update of Inverted List of Text Document Retrieval. In *Proceedings of 1994 ACM SIGMOD*, May 1994.
- [147] Anthony Tomasic, Luis Gravano, Calvin Lue, Peter Schwarz, and Laura Haas. Data Structures for Efficient Broker Implementation. *ACM Transactions on Information Systems*, 15(3):223–253, 1997.
- [148] C.J. Van Rijsbergen. *Information Retrieval*. Butterworths, 1979. Available at <http://www.dcs.gla.ac.uk/Keith/Preface.html>.
- [149] Freeweb web site. <http://freenet.sourceforge.net>.
- [150] H. Williams and J. Zobel. Compressing integers for fast file access. *Computer Journal*, 42(3):193–201, 1999.
- [151] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes – Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishing, San Francisco, second edition edition, 1999.
- [152] Y. Xie and D. O’Hallaron. Locality in search engine queries and its implications for caching. In *Proceedings of IEEE INFOCOM 2002, The 21st Annual Joint Conference of the IEEE Computer and Communications Societies*, 2002.
- [153] Xu, Jinxi, and W.B. Croft. Effective Retrieval with Distributed Collections. In *Proceedings of SIGIR98 conference*, Melbourne, Australia, August 1998.
- [154] Jinxi Xu and W. Bruce Croft. Cluster-Based Language Models for Distributed Retrieval. In *Research and Development in Information Retrieval*, pages 254–261, 1999.
- [155] B. Yang and Garcia-Molina. Comparing Hybrid Peer-to-Peer Systems. In *Proceedings of the 27th VLDB*, September 2001.
- [156] Budi Yuwono and Dik Lun Lee. Server Ranking for Distributed Text Retrieval Systems on the Internet. In *Database Systems for Advanced Applications*, pages 41–50, 1997.
- [157] B. Zhao, J. Kubiawicz, and A. Joseph. Tapestry: an Infrastructure for Fault-Tolerant Wide-Area Location and Routing. Technical Report UCB/CSD–01–1141, Computer Science Division, U. C. Berkeley, April 2001. Available at <http://www.cs.berkeley.edu/ravenben/publications/CSD–01–1141.pdf>.

- [158] G. K. Zipf. *Human Behavior and the Principle of Least-Effort*. Addison-Wesley, Cambridge, MA, 1949.