SIDRA: a Flexible Web Search System

Miguel Costa

DI–FCUL TR–2004–17

December 2004

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

SIDRA: a Flexible Web Search System

Miguel Ângelo Leal da Costa

Dissertação submetida para obtenção do grau de

MESTRE EM INFORMÁTICA

Orientador:

Mário Jorge Costa Gaspar da Silva

Júri:

Helena Isabel de Jesus Galhardas

Luís Manuel Pinto da Rocha Carriço

Nuno Fuentecilla Maia Ferreira Neves

Setembro de 2004

SIDRA: a Flexible Web Search System

Miguel Ângelo Leal da Costa

Dissertação submetida para obtenção do grau de
MESTRE EM INFORMÁTICA

pela

Faculdade de Ciências da Universidade de Lisboa

Departamento de Informática

Orientador:

Mário Jorge Costa Gaspar da Silva

Júri:

Helena Isabel de Jesus Galhardas

Luís Manuel Pinto da Rocha Carriço

Nuno Fuentecilla Maia Ferreira Neves

Setembro de 2004

Abstract

Sidra is a new indexing, searching and ranking system for Web contents. It has a flexible, parallel, distributed and scalable architecture. Sidra maintains several data structures that provide multiple access methods to different data dimensions, giving it the capability to select results reflecting search contexts. Its design addresses current challenges of Web search engines: high performance, short searching and indexing times, good quality of results, scalability and high service availability.

KEY-WORDS: Web, search engines, indexing, ranking, information retrieval.

Resumo

O Sidra é um novo sistema de indexação, pesquisa e ordenação de conteúdos da Web. Possui uma arquitectura flexível, paralela, distribuída e escalável. Contém várias estruturas de dados para acesso a diferentes dimensões de dados, o que lhe permite seleccionar resultados reflectindo o contexto das pesquisas. O seu desenho procura responder aos desafios actuais dos motores de pesquisa para a Web: alto desempenho, tempos de pesquisa e indexação reduzidos, boa qualidade dos resultados, escalabilidade e alta disponibilidade de serviço.

PALAVRAS-CHAVE: Web, motores de pesquisa, indexação, ordenação, recuperação de informação.

Agradecimentos

Esta tese de mestrado não teria sido possível sem a ajuda e apoio de algumas pessoas, das quais muitas vezes me tive de privar e de não lhes dar a atenção merecida. Nesta secção tenho a oportunidade de lhes fazer um pequeno agradecimento por escrito, demonstrando assim um pouco do sentimento de gratidão que nutro por elas.

Em primeiro lugar, quero agradecer ao meu professor e orientador Mário Silva, pela sua sábia orientação e ajuda na elaboração desta tese. Não posso também deixar de lhe agradecer a oportunidade que me deu no grupo XLDB, onde muito me ensinou.

Esta tese foi desenvolvida com a cooperação dos meus colegas do grupo XLDB, onde o seu apoio e conhecimento foram fundamentais para a sua conclusão. Obrigado a todos, em especial àqueles que mais directamente trabalharam comigo: João Campos, Daniel Gomes, Norman Noronha e Bruno Martins.

Um obrigado também ao Renato Torres e ao Marcirio Chaves pelos seus preciosos comentários na revisão desta tese.

Um especial agradecimento à Fundação da Faculdade de Ciências da Universidade de Lisboa (FFCUL) e ao Instituto de Ciência Aplicada e Tecnologia (ICAT), pelo financiamento da minha dissertação de mestrado.

Para os meus amigos que me "arrancavam" de casa e me mostravam que a vida é muito mais do que uma tese, um abraço sentido, em especial para o Rui

Grilo e para o Bruno Moutinho.

Ao meu grande amor ... Ana, tu foste a motivação e a força para acabar esta tese quando as perdi; tu foste o porto que me abrigou nos dias de tempestade e o vento que me levou ao fim do percurso nos dias de sol; tu foste tudo e muito mais. Jamais terei palavras para te agradecer.

Aos meus pais, Armando e Dália, a minha eterna gratidão por todo o vosso amor e pela vossa força. Foi graças a todo o vosso apoio que tive a oportunidade de estudar o que sempre gostei e de concluir esta tese nessa mesma área. É também por vocês que sinto a maior alegria ao concluir esta tese, podendo retribuir um pouco do orgulho que tenho em vós.

Lisboa, Setembro de 2004

Miguel Ângelo Leal da Costa

Dedico esta dissertação aos meus pais e à minha princesa.

Contents

Contents **i**

List of Figures **v**

List of Tables **vii**

1 Introduction **1**

1.1 Objectives 3

1.2 Results 4

1.3 Methodology 6

1.4 Organization 7

2 Concepts and Related Work **9**

2.1 Concepts 9

2.1.1 Retrieval Models 10

2.1.2 Indexes 11

2.2 IR Architectures 14

2.2.1 IR and Database Systems 15

2.2.2 Web Search Engines 16

2.3 Conclusion 16

3 Searching System **19**

3.1 Architecture 20

3.2 Distributed Query Processing 22

3.2.1 Distributed Joins 22

3.2.2 Distributed Query Plans 24

3.3 Indexes Design 25

3.4 Results 27

3.4.1 Test Environment 28

3.4.2 Tests 31

3.5 Results Analysis 37

3.5.1 Comparative Results 39

3.6 Conclusion 41

4 Ranking System **43**

4.1 Ranking Framework 44

4.1.1 Functionality 45

4.1.2 Multi-dimensionality 46

4.1.3 Evaluation 47

4.2 Optimizing Ranking Calculation 51

4.2.1 Preliminaries 51

4.2.2 Reducing Ranking Calculation 52

4.2.3 Searching for a Good Solution 53

4.2.4 Results 58

4.2.5 Results Analysis 61

4.2.6 Number of Query Matches 63

4.3 Conclusion 64

5 Indexing System **67**

5.1 Centralized Indexing Algorithm 68

5.2 Sidra's Distributed Indexing Algorithm 69

5.2.1 Generating Runs 70

5.2.2 Merging Runs 71

5.2.3 Building Inverted Files 72

5.2.4 Fault Tolerance 73

5.3 Results 75

5.3.1 Testbed 76

5.3.2 Tests and Analysis 76

5.3.3 Comparative Results 77

5.4 Index Updates 79

5.5 Conclusion 81

6 Conclusions and Future Work **83**

6.1 Future Work 84

Bibliography **87**

List of Figures

1.1 Tumba!'s architecture. 2
1.2 Layout of tumba!'s results. 5
1.3 Incremental model adopted on the first release of the system. . . . 6
1.4 Spiral model adopted on the following releases of the system. . . . 6

2.1 Inverted file structure. 12
2.2 Major proposals to partition inverted files. 13

3.1 One possible configuration of Sidra's architecture. 21
3.2 Design of Sidra's indexes. 26
3.3 Response times varying the number of computers on configurations. 37

4.1 extPageRank distribution. 55
4.2 URL weight distribution. 55
4.3 $\widetilde{n}$ as a function of Stat with the index sorted by extPageRank. . . . 58
4.4 $\widetilde{n}$ as a function of Stat with the index sorted by URL weight. . . . 59
4.5 $\widetilde{n}$ as a function of TA-Adapt with the index sorted by extPageRank. 60
4.6 $\widetilde{n}$ as a function of TA-Adapt with the index sorted by URL weight. 60
4.7 Reductions using several algorithm combinations. 61
4.8 Reductions using several coefficients c in the *rank* function. 61

5.1 Sidra's distributed architecture to create indexes. 70

5.2 Sidra's distributed architecture to create indexes with fault tolerance. 74
5.3 Times to index the 3.2M Web collection. 77

List of Tables

1.1 Summary of results. 4

3.1 Some of the rules of the plan generator. 24
3.2 Test parameters 28
3.3 Response times for the baseline tests. 32
3.4 Response times for tests with an avg of 5 and 10 terms per query. . 33
3.5 Response times for tests with high frequency terms in collection. . 33
3.6 Response times for tests with 100 and 1000 results returned. . . . 34
3.7 Response times for tests with the 44K collection. 35
3.8 Response times for tests with 2 and 4 computers (1st configuration) 36
3.9 Response times for tests with 2 and 4 computers (2nd configuration) 38
3.10 Distributed systems to search large scale collections. 40

4.1 Result pages seen by users in Web search engines. 49
4.2 Weights given to each class of *imp* functions. 55

5.1 Distributed systems to build large scale inverted files. 78

Chapter 1

Introduction

The World Wide Web is an enormous repository of digital data available on computers spread all over the world [13]. Information is now at the distance of a click, but first it is necessary to find it. Web search engines are the tools usually used for this purpose, receiving hundreds of millions of queries per day. There are few systems with so high requisites. To support them, Web search engines integrate a large variety of techniques from the high level architectural software design to the low level hardware configurations. Some of these configurations have computational power comparable to world's fastest super computers [11]. Still, having all the techniques used to find information on the Web as fast as possible, Web search engines aren't perfect and face many challenges. The origin of these challenges is mostly due to the large dimension and fast expansion of the Web, making it difficult for Web search engines to keep up offering the same response times and quality of results.

Tumba! is a search engine for the Portuguese Web which is available as a public service since 2002 [75]. It came across with these challenges when it grew from an academic project indexing tens of thousands of documents, to a large project that indexes the Portuguese Web. It is composed by several systems, each

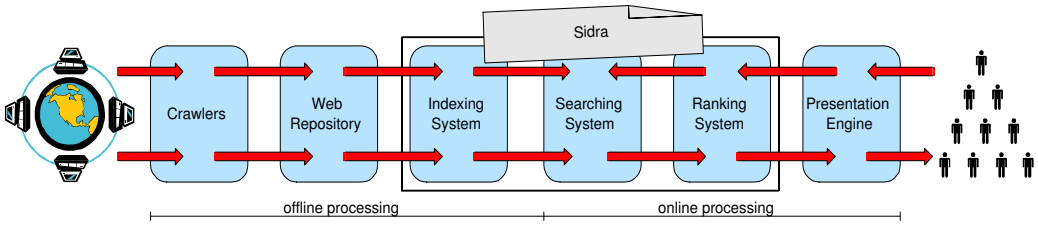


Figure 1.1: Tumba!'s architecture.

one interacting to identify the documents from a Web data collection that best matches users' needs. Tumba! has six systems, as in most search engines architectures (see Figure 1.1). The data flows from the Web through tumba!'s systems, which successively transform it until it is sent to the user:

Crawlers: given an URL list (seeds), collect the respective documents from the Web [36, 34]. Crawlers parse and extract URLs from each collected document, which will be used to collect new documents. These steps are performed recursively until a stop condition is met.

Web Repository: stores the data collected from crawlers in Versus, a distributed repository of Web documents and associated metadata [24, 35].

Indexing system: creates indexes over the stored information in the Web Repository to speed up the search process.

Searching system: when a query is received, uses the indexes to lookup the set of documents matching the query.

Ranking system: computes, for each document d returned by the searching system, the relevance of d to the submitted query, using a set of heuristics. Then, it returns the documents sorted by these relevance values [28].

Presentation Engine: formats the result sets received from the ranking engine for users' access platforms, such as Web browsers or WAP phones.

This dissertation describes **Sidra**, a system that implements the indexing, searching and ranking system functions of tumba!. These components work in tandem, each one taking advantage of this symbiosis to optimize Web searches. Its design focused on satisfying tumba!'s requisites, which are not supported by available solutions.

- Commercial indexing and searching systems are slow and do not scale due to the organization of the index structures and core parts, which can not be optimized. A previous version of tumba! built on top of Oracle InterMedia suffered from these problems [4].
- Open source systems available (e.g. Jakarta Lucene [3]) are centralized and not designed for Web ranking. Changing them to support tumba!'s requisites would be comparable to build a new system from scratch.

1.1 Objectives

Sidra was developed to provide search capabilities for the tumba! Web search engine, meeting its demanding requirements. Specifically, Sidra should support five main requisites:

searching times: to offer high performance response times, even with high workloads (95% of the response times inferior to 1 second).

indexing times: to provide high performance indexing times, essential to refresh indexes of highly volatile Web data (index the Portuguese Web, 3.2 million documents, in less than a week).

Requisites	*Values*
searching times	100 millisec with 50 requests/sec over 156.8 GB
indexing times	56.38 hours to index 313.6 GB
quality of results	comparable to Google; ranking framework implemented
scalability	searching and indexing times scale linearly
service availability	provides fault tolerance mechanisms

Table 1.1: Summary of results.

quality of results: to build a framework to develop, test, evaluate and use ranking algorithms on tumba!, aimed at improving accuracy to levels similar or better than the ones produced by current Web search engines (compared through ranking metrics).

scalability: to scale the searching and indexing times as the indexed Web collections grow.

service availability: to provide fault tolerance mechanisms, enabling Web search engines response even if some of its components crash.

1.2 Results

Sidra was designed to support the objectives outlined above. For that purpose, Sidra has a flexible architecture that enables the deployment of different configurations to respond to the needs of tumba!, while also supporting load balancing and fault tolerance.

The scalability and performance of the Sidra implementation was evaluated over a realistic set of queries extracted from tumba!'s logs, applied with different workloads to multiple Sidra configurations. Measurements show that the searching and indexing times of the system scale linearly. Results also show that these times are comparable, some times ever better, than those obtained by similar state-

Figure 1.2: Layout of tumba!'s results.

of-the-art systems. Sidra's tests over a collection with 3.2 million Web documents replicated 2 times (156.8 GB), demonstrate that the system could sustain response times of 100 milliseconds with a workload of 50 requests per second, using a cluster configuration with 4 computers. The indexing of the same collection replicated 4 times (313.6 GB), took 56.38 hours using the same cluster of computers. Table 1.1 summarizes the main results achieved with Sidra.

Presently, Sidra is in operation in tumba!, supporting the parallel processing of keyword searches, Boolean operators and phrase searches. It responds to more than 20 thousand queries a day over a collection of 3.2 million Web documents. Figure 1.2 depicts tumba!'s results page, which presents the data produced by Sidra in response to users' queries.

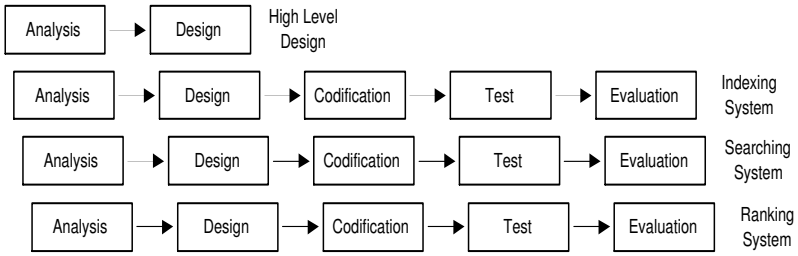


Figure 1.3: Incremental model adopted on the development of the first release of the system.

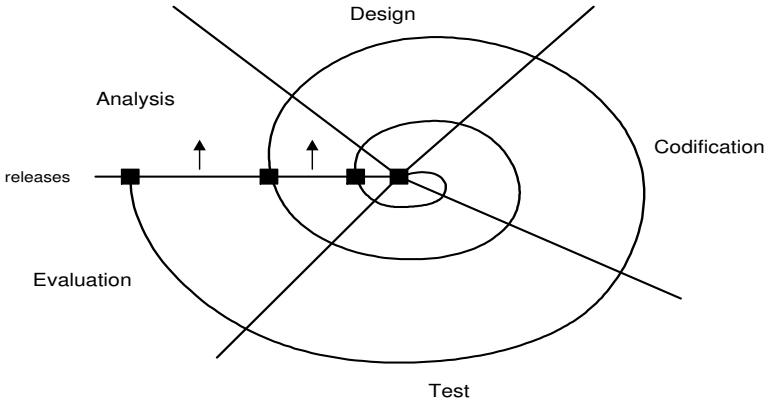


Figure 1.4: Spiral model adopted on the following releases of the system.

1.3 Methodology

Sidra was developed in incremental steps. First, a high-level design of Sidra was completed. Then, its three systems were developed in the following order: indexing system, searching system and ranking system. Each of these systems adopted the same incremental development model [64]. In each increment, each system or component followed the approach of the linear sequential model, passing through five sequential stages: analysis, design, codification, test and evaluation. Figure 1.3 represents this process.

At the end of the construction and integration of all systems, the spiral model

[64] was adopted (see Figure 1.4). Sidra was developed in a series of incremental releases, each one adding new functionalities to produce a more complete version of the system and more adjustable to the users' needs. The end of the incremental model corresponds to the center of the spiral, which indicates the conclusion of the first release. Several other releases were built, driven by the continuous feedback of tumba!'s users for about 9 months. New releases produced over time offered better searching and indexing times, more accurate results, new search operators (phrase and within site search) and easier to use configuration interfaces.

1.4 Organization

This thesis is organized in 6 chapters. Next chapter presents concepts and related works. Chapters 3, 4 and 5 detail the architecture, implementation and validation of Sidra's searching, ranking and indexing systems, respectively. This organization does not follow the flow of data through Sidra's systems, because the indexing system was developed to optimize the generation of the data structures used by the searching and ranking systems. Therefore, it is easier to start with the last two systems and only then present the indexing system. Chapter 6 gives the final conclusions.

Chapter 2

Concepts and Related Work

This Chapter presents the concepts and then the architectural paradigms of text retrieval systems. Some of these have been used in the high level design of Sidra. In the conclusion, Sidra's main architectural decisions are discussed.

2.1 Concepts

Information Retrieval (IR) studies the search of documents containing needed information with the help of computational resources. It is a broad interdisciplinary field that draws on many other disciplines, such as cognitive psychology, computer science, user interfaces, data visualization, human information behavior, linguistics and librarianship. IR usually deals with matching natural language text documents with user's queries following a retrieval model. These queries are submitted to a IR system which uses indexes to speed up the matching between queries and documents. IR retrieval models and indexing techniques are described in this Section.

2.1.1 Retrieval Models

A retrieval model is an abstract representation which describes the human and computational processes involved in IR [9]. The retrieval models most extensively used are the Boolean model, the Vector Space model, and a combination of both.

Boolean Model

The Boolean retrieval model represents the way information is searched in typical Web search engines. It is based on set theory and Boolean algebra. Each query is composed by a set of terms, each one representing a set of documents containing the term. The terms are connected with logical operators AND, OR and NOT, and the document sets are, respectively, intersected, united or excluded to produce the final result. The resulting documents are considered relevant with an equal relevance to the query, so they are returned to the user in no particular order. If many documents are returned, its is difficult for the user to find the most relevant to the search.

Vector Space Model

The Vector Space model represents documents and queries as vectors of term weights. For a submitted query q and each document d of the collection, the Vector Space model assigns weights to the terms of q and d, defining a query vector $\overrightarrow{q}$ and a document vector $\overrightarrow{d}$ of weights, respectively. The number of dimensions is equal to the number of distinct terms in the collection. The similarity between q and d is measured as the inner product between the vectors $\overrightarrow{q}$ and $\overrightarrow{d}$, usually normalized by the cosine of the angle between these two vectors [90]. The documents are then returned sorted by the computed similarity.

Boolean + Vector Space model

Modern IR systems use a mix of the Boolean and Vector Space models. The interaction with the user is the same as in the Boolean model, where the user can submit Boolean expressions of terms as queries. Then, the documents in the result set are weighted using some algorithm based on the Vector Space model. This combination enables users to use expressive logic operations of the Boolean model, with the partial matching of the Vector Space model, which offers better retrieval precision and outputs results sorted by similarity.

2.1.2 Indexes

The potential large size of a full text collection demands specialized techniques for efficient IR. These techniques include index structures with their intrinsic problems and virtues that influence architectures and the processing between their components.

Index Data Structures

There are two major index data structures for efficient IR of large collections of documents: *inverted files* and *signature files*. Evaluation results of these structures all point in the same direction. Inverted files enable the best compression ratio and response times [92, 90, 9].

An inverted file, also known as *posting file*, is a word index similar to the word index of a book. The keys are the *terms* that occur on the collection of documents. The set of all terms in the collection is called a *lexicon*. Each term points to a list of identifiers of the documents where the term occurs. This list is called a *posting list* also known as *inverted list*. Each of these identifiers, usually with information associated as the term frequency into the document, is called a *posting*. Figure 2.1

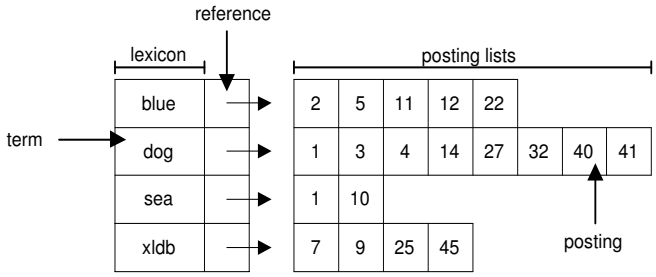


Figure 2.1: Inverted file structure.

represents the structure of an inverted file.

The lexicon is normally available in memory during runtime to enable fast searches. The posting lists are accessed from disk due to their size being larger than the memory available. To find the documents that match a query of terms, each term is searched on the lexicon and the posting list it points to is fetched. The result is the intersection of the identifiers of the posting lists.

Inverted Indexes Partitioning

Web collections have reached sizes of billions of documents and still continue to grow. It is impossible to build, store and search efficiently indexes of these collections with only one computer. A scalable approach requires the partitioning of the index by several computers, to enable parallel generation and access. This partitioning strongly influences the performance and scalability of a system of this nature. There are two major alternatives for partitioning an inverted index for an IR distributed architecture: the *local partition*, also known as vertical partition, and the *global partition*, also known as horizontal partition. Figure 2.2 represents both structures.

Using a local partition, each computer builds a complete inverted index over a disjoint set of documents of the collection and only answers queries on these doc-

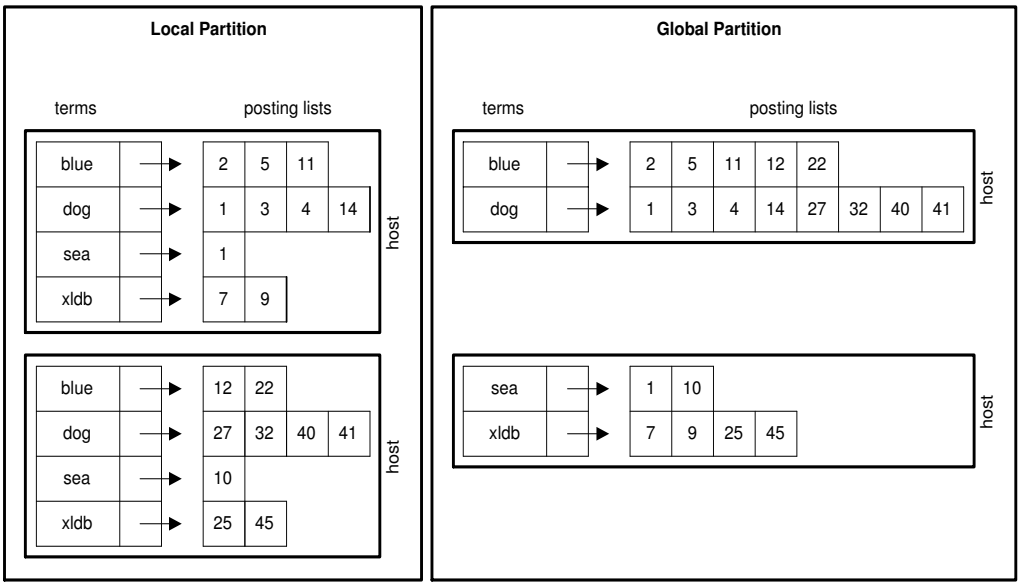


Figure 2.2: Major proposals to partition inverted files.

uments. For each query q received, a Broker resends q to all computers holding a partition of the index, denoted QueryServers, requesting the documents that match q. Each term of q originates in all QueryServers a disk access and a disk transfer of a fraction of the posting list. The results are then sent to the Broker, which merges them. This partition has two advantages. First, the local organization enables good parallelism, because all QueryServers are devoted to the execution of each query. Second, all information about a document is available locally, so it is possible to compute the ranking of the documents in parallel without any interaction between computers. As disadvantage, there is a high concurrency of disk operations during high workloads, because all computers always receive a request for each query.

In global partition, the inverted index contains a disjoint set of all the terms of the collection and associated posting lists. For each query q received, each of the QueryServers with query terms in its index, and only these, receives a new

query. Each QueryServer, for each received term, performs a disk access and a disk transfer of the posting list. Results are sent back to the Broker, which merges them. The advantage of the global partition is the smaller number of requests and disk operations received per computer, enabling more concurrency. A disadvantage is the distribution of information about one document, so it is necessary that the Broker first merges all the information from the QueryServers involved in q, before matching and ranking the results.

The choice between both structures depends of the network bandwidth, the disk performance and the implementation easiness. A query search using local partition is easy to implement because each QueryServer can process a part of the request independently. With the global partition, the Broker has to interact several times with the QueryServers until a request is processed. Performance results indicate that a global partition outperforms a local partition in the presence of fast communication channels (above 100 Mbits/sec), being increasingly better as the bandwidth increases [83, 47, 48, 67].

2.2 IR Architectures

Initially, IR architectures were dominated by powerful multiprocessor mainframes [80, 81, 33, 10]. However, mainframes were expensive and do not scale. An alternative are distributed share-nothing architectures, connected by fast networks. Parallel processing of partitioned data allows systems to take profit of the aggregated bandwidth of multiple disks and CPUs, providing impressive scalability.

Burkowski reported a simulation study where performance could approach linear speedup on a share-nothing architecture up to 16 servers [21]. Hawking designed and implemented a parallel distributed IR system called PADRE97 [43]. Its architecture includes one intermediary computer, called a Broker, that broad-

casts user requests to all the computers with a subset of the documents indexed (QueryServers). Hawking showed that query processing presented linear scalability up to 9 computers over a collection with 10.2 GB of text per QueryServer. However, he focused on the scaleup of the response time of a single query.

Inquery is a centralized retrieval engine, whose retrieval model is based on a Bayesian inference network [23]. Cahoon and McKinley developed a simulation model to study distributed share-nothing architectures using Inquery servers [22]. They simulated configurations with up to 128 servers, each containing a 1 GB text collection. The architecture has Clients searching over multiple text collection simultaneously, selecting the collections more relevant to each query. Clients send the queries to a Broker, which dispatches them to the Inquery servers. Their simulation model, validated with only one Inquery server, predicted that performance increases with up to 32 servers.

2.2.1 IR and Database Systems

The knowledge originated in the architectural design of parallel Database Management Systems (DBMSs) has been incorporated in IR architectures and vice versa over the years. Both use parallelism to scale query processing in a similar way [29]. However, there are differences, resulting from the nature of the managed data. IR systems search over documents written in natural language, instead of structured data as in DBMSs.

Some IR systems have been developed on top of relational DBMSs, using functionalities like concurrency and recovery control to reduce development time. SIRE, is a Scalable Information Retrieval Engine, which maps the inverted index on relational database tables [32]. PowerDB-IR is a software layer for database clusters that provides a scalable infrastructure for IR [38]. It does the same mapping to take profit of the highly parallelization of relational operators.

2.2.2 Web Search Engines

The major differences between Web IR architectures and traditional IR architectures result from differences on the collections indexed. Web collections tend to be much larger and more dynamic. Most of their documents are unstructured and present an enormous heterogeneity of formats.

There are two Web search engines with architectures closely related to Sidra. The Google architecture contains several QueryServers, each one responding to a subset of documents (local partition) [15, 11]. Each QueryServer has several replicas. When a query is performed, a Broker receives and dispatches it for one of the replicas of each QueryServer, which they process it in parallel. After receiving the results, the Broker merges them. Meanwhile, the other replicas are free to receive other requests. Thus, Google reduces the high concurrency of requests for each QueryServer, typical in systems with a local partition organization. It is also interesting to see that beyond the intranet replication of the components, the Google system is completely replicated worldwide and the DNS maps `www.google.com` to a particular IP address according to the location of the search. In this simple way, Google supports a much higher workload. The second architecture is from the FAST search engine, also similar to the one adopted by Google [71]. The difference is that it has multi-levels of Brokers, organized in a tree-like structure, to ensure that the Broker does not become a bottleneck.

2.3 Conclusion

The previous sections presented the main concepts and alternative configurations that have been used up to now for designing IR systems. Sidra can now be positioned in this design space.

Sidra's retrieval model is a combination of the Boolean model, which enables

the use of expressive logic operations between query terms, with the partial matching of the Vector Space model that offers better retrieval precision and results sorted by similarity.

Sidra's index structures are inverted files, the fastest and more compressive search structure.

Sidra has a share-nothing architecture, composed by inexpensive computers connected by a high speed network. Parallel processing results from the partitioning of independent data among available computers, and also from the global partitioning of indexes, which is said to achieve the best performance in the presence of fast communication channels (above 100 Mbits/sec).

Sidra partitions the indexes as relational DBMSs partition relational database tables (global partition), and uses similar operators (intersection, union and difference) in query processing to achieve the same high level of parallelism as relational DBMSs.

From Web search engines architectures, Sidra inherits the design choice of replicating components to prevent bottlenecks, implement fault tolerance, provide load balancing mechanisms, and improve search performance.

Most of the IR systems described, have a three-tiered architecture. Sidra adopted a similar architecture, composed by *Clients* providing the interface between the user and the system, and *Brokers* dispatching requests to *QueryServers* that index part of the data. This architecture is explained in detail in the next Chapter.

Chapter 3

Searching System

Existing Web search engines are the crystal balls of our times. They are required to serve millions of users while providing the best possible results for any query in a fraction of a second. However, the more collections grow, the more difficult it is for Web search engines to offer good results. State of the art Web search engines, such as Google, have an indexing structure that provides access methods where a unique ranking dimension is used to search Web pages [15]. On the other hand, improvements in ranking algorithms enrich their heuristics with the semantic information available from the documents and queries [70, 42].

In Web environments, user preferences change over time and the same query means different things to different users. As mobile devices become a common interface to query these systems, there is a need to provide adaptive responses. For instance, location dependent queries will provide different results to the same query, depending on user preferences explicitly provided or inferred from their computing environment.

Sidra supports several distributed indexing data structures that can be organized by different importance criteria and may be selected to find matches based on the context-data associated to queries. To provide high-scalability and sub-

second response times in such multi-dimensional environment, relevance of results matching a query needs to be evaluated using all these indexes. This approach significantly increases the storage requirements, but preserves the sub-second response times currently demanded by search engines' users.

This Chapter details in Section 3.1 Sidra's searching architecture, and the distributed and parallel query processing in Section 3.2. The design of the index data structures is presented in Section 3.3. Results of the performance and scalability of the system are presented in Section 3.4. Section 3.5 analyzes these results and compares them with similar results from other systems. Section 3.6 presents the conclusion.

3.1 Architecture

Sidra's searching architecture has three types of components: QueryServers, Brokers and Clients. Different indexes are organized and partitioned by different criteria. Each index partition is managed by a QueryServer, responsible for matching queries received with a list of Sidra document identifiers (sids), and associated information. Figure 3.1 depicts a possible configuration of Sidra's architecture, with three different types of indexes: term-documents, location-documents and topic-documents. These, enable document searches on up to three search dimensions. A possible query supported by this system would be: term=*mustang* AND topic=*animals* AND location=*Portugal*. The system would then match the Portuguese documents related with mustang horses. Other search dimensions can be added.

Sidra's indexes are partitioned by multiple QueryServers on a global partition scheme, allowing fast searches on different search dimensions in parallel (partition parallelism). Posting lists are distributed across all QueryServers on a range of

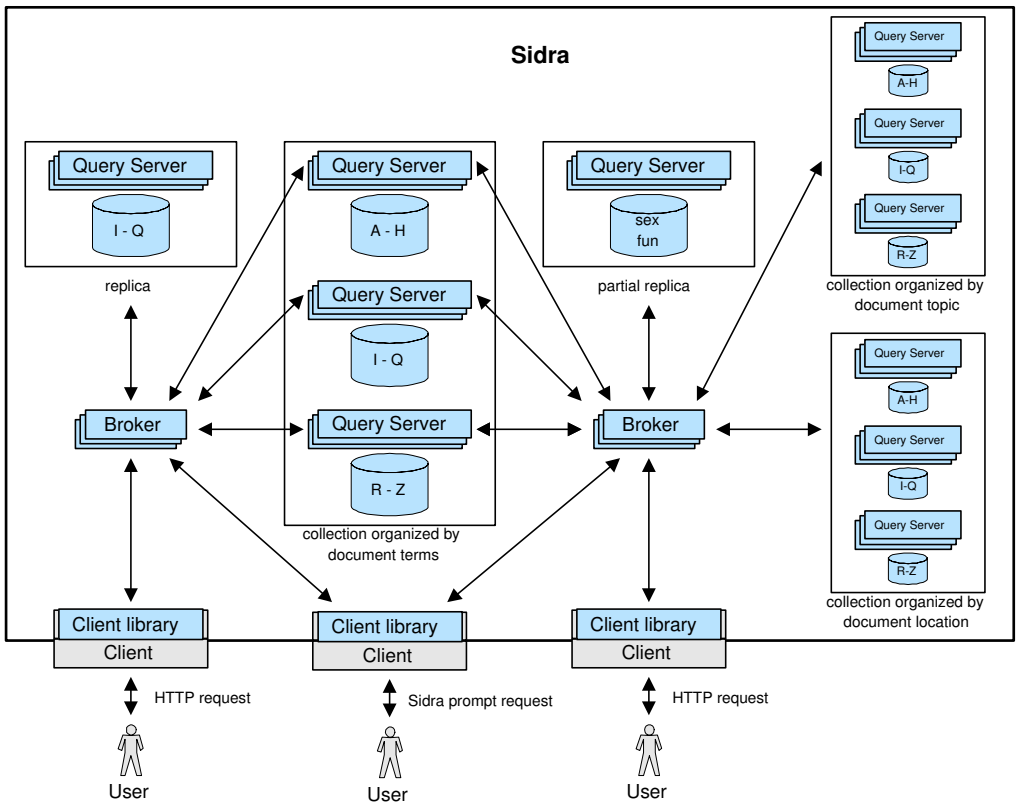


Figure 3.1: One possible configuration of Sidra's architecture.

index terms.

Brokers receive assembled queries from Clients in the form of search expressions. They use the information on a shared catalog with the distribution of the indexes and the location of the QueryServers to dispatch sub-queries to selected QueryServers. Each sub-query requests matching documents and all the associated information. Then, Brokers evaluate and intersect the results received in parallel from the QueryServers as they are being produced. This provides a pipeline parallelism that enhances query performance. Finally, documents are ranked according to the indexes chosen by search criteria.

Sidra's Clients are lightweight applications that connect users to Sidra through

a programming interface. Clients then format Brokers responses, transforming the data for presentation on user devices.

Sidra has a modular and flexible architecture, which enables the deployment of different configurations to respond to the needs of different Web search engines. At a lower level, the system can grow incrementally, adding memory, storage and CPU with the addition of inexpensive computers. The performance of these computers communicating by messages over a fast network can exceed mainframe performance by the same price and achieve the computational power of supercomputers (see for instance, `http://www.top500.org`). At a higher level, components can be incremented, replicated or partially replicated as the data and query workloads increase. As replicating a full QueryServer can represent a significant amount of storage resources, Sidra's architecture enables the creation of partial replicas of QueryServers for the more frequently accessed entries.

Sidra's replication functions were extended, developing a mechanism to perform load balancing by distributing the requests to the less loaded. Along with these capabilities, requester components can detect, during search time, components that are not responding or exhibiting high response times. These requests can then be redirected to a replica, providing a fault tolerance mechanism. All components also have software watchdogs that monitor and restart them once a fault is detected.

3.2 Distributed Query Processing

3.2.1 Distributed Joins

Sidra's indexes are physically divided using a global partition. This means that each term of an inverted index has the associated posting list on a single QueryServer. When a query is requested and all query terms are available at the same

QueryServer, the result is the intersection of the posting lists of these terms. For instance, for the query composed by the terms *java* and *coffee*, the posting lists of these two terms are fetched from the term-documents index and intersected.

The intersection of posting lists among distributed QueryServers is similar to computing an equijoin in distributed relational databases. At least one of the posting lists or its representation must be sent to a remote computer. Techniques to compute distributed joins, such as *fetch as needed* or *ship to one site* where all tuples are sent from one site to another are inefficient [65]. Sidra uses an equivalent to semijoins, sending the QueryServers only a projection of the sids for the Broker who performs the join. After joining the sids, a Broker requests the ranking information of the documents identified by these sids to the same QueryServers. The documents are then ranked according to the selected search dimensions.

The semijoin could transmit large amounts of sids to the Broker. To reduce network communication, three improvements were considered:

1. Tomasic and Garcia-Molina tested a Broker dispatching queries for the QueryServer that would receive the term with the shortest posting list or the largest number of terms [83]. The QueryServer resends the query with its matching sids to the other QueryServers, which intersect these with their own matching sids. However, this strategy presents worse performance than the semijoin used because it reduces parallelism. It creates a dependency on the query processing workflow, compelling the QueryServers to wait for the QueryServer that first receives a query from the Broker.

2. instead of transmitting sids, bloom filters may be used to send a hash-based data structure that summarizes membership in a set of sids [14, 57, 66]. Sidra does not presently use this kind of optimization.

search expression		distributed execution tree		
$A \cap B \cap C \cap D$	$\equiv$	$QueryServer_1$ $A \cap B$	Broker $\cap$	$QueryServer_2$ $C \cap D$
$A \cap B \cap C - D$	$\equiv$	$QueryServer_1$ $A \cap B$	Broker $\cap$	$QueryServer_2$ $C - D$
$A \cap B - C - D$	$\equiv$	$QueryServer_1$ $A \cap B$	Broker $-$	$QueryServer_2$ $C \cup D$

Table 3.1: Some of the rules of the plan generator. A, B, C and D are sets of sids from the posting lists of four searchable terms. $QueryServer_1$ contains posting lists with the sets A and B. $QueryServer_2$ contains posting lists with the sets C and D.

3. semijoins can be performed incrementally. Sidra's Broker requests segments of sids from QueryServers until it has the top k more relevant to the query. This optimization is key to Sidra's ranking system, which is detailed further in Chapter 4.

3.2.2 Distributed Query Plans

Sidra's search expression operators are highly parallizable. They were implemented adopting design patterns that have been initially devised for parallel relational database operators. In particular, these operators have many similarities with the sort-merge join algorithm when the relations are already sorted on the join columns. A simple query plan generator was designed, that runs in Brokers. It follows the rules of Table 3.1 to generate distribute execution trees for each search expression.

Processing queries according to these distributed execution trees, enables Que-

ryServers to process disjunctive parts of the query in parallel (partition parallelism). During the the logic operations in Sidra, Brokers read streams of sids from QueryServers to merge the sids as needed and as they become available from the QueryServers. Brokers and QueryServers are in this way processing a query in a pipeline parallelism.

3.3 Indexes Design

In IR systems like Sidra, disk transfers and specially random disk seeks, tend to limit systems' performance. Sidra's indexes were designed to reduce disk seeks and favor sequential access to disks. This has a major importance in the design of a high performance searching system. Presently, a normal data transfer rate in an ATA disk takes between 500 and 800 Mbits/sec, while a random disk seek takes between 8 and 11 milliseconds (see for instance `http://www.seagate.com`). As Sidra uses many indexes, if the index structures were not optimized, much more than one second, the response time set as objective, would be spent in i/o alone.

Figure 3.2 represents the structures of some of the Sidra's indexes used to process a query. Indexes were designed to be accessed for each query term only once, when index records have fixed length (e.g. as hits index), or two times, when index records have variable length (e.g. as positions index). All indexes are inverted files where each term on the lexicon has associated a list of records. Records contain data for each pair <term,document>, different between indexes. For instance, a record in the term-documents index contains a posting, while a record in the hits index contains the ranking values related with a term on the document. All records assigned to the same pair <term,document> on the different indexes, are stored on the same *order* in the lists of records. For instance, in the

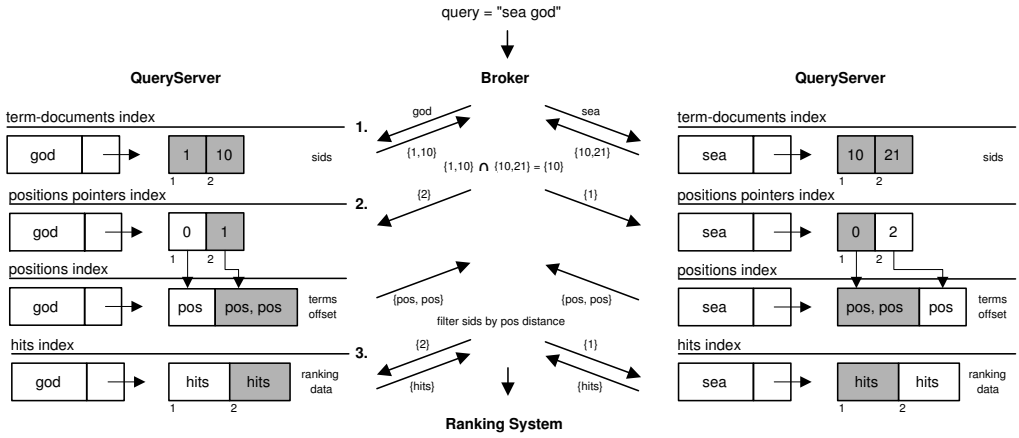


Figure 3.2: Design of Sidra's indexes.

example of Figure 3.2, all records of the term *god* and sid *10* occur on the second position. Knowing the record size of each index, it is only necessary to know the order of a sid on the posting list, to fetch from the other indexes all the data associated to that sid.

Indexes with variable length records need an intermediary step. The order is not enough to calculate the offset of a record on the list. The solution is to build a new index of pointers with fixed length (e.g. as the positions pointers index), containing the offset of the variable length records of the index pointed (e.g. as the positions index).

Figure 3.2 depicts the use of these structures during the processing of the query phrase *sea god*.

1. the Broker receives a query from the Client, and uses the catalog with the distribution of the indexes to dispatch each term for the QueryServer which indexed it. Each QueryServer fetches the respective posting list and sends it to the Broker, which intersects them. In the example, the matching set is composed by sid 10, the only document that contains these two terms.

2. as the query is a phrase search, it is necessary to filter the sids that do not have the terms adjacent. For that, the Broker requests the positions of these two terms for the document with sid 10. The Broker does not send the sid, but the order of the sid on the posting list received. This order enables QueryServers to access to the positions pointers index, which has an offset for the record that contains the positions of sid 10 on the positions index. QueryServers send these positions to the Broker, which filters the sids that does not have the query terms adjacent. Note that this step it is only necessary when the query is a phrase search.

3. in the end, the Broker requests the ranking information of the resulting sids, sending again their order. The Broker then receives all the ranking data of the matching sids, which are passed to the ranking system.

Sidra's indexes were also designed to be modular, enabling QueryServers to search them independently of the information indexed. For instance, an index of domains was built for tumba!, having associated to each domain a list with all the documents pertaining to it. QueryServers and Brokers use it exactly in the same way as term-documents indexes. The only difference is that the Broker needs to be informed about which query terms represent domains to generate proper query plans. In the tumba! user interface, these terms are prefixed with the string "site:".

3.4 Results

In this Section, it is presented the performance and scalability results achieved during the experiments conducted to validate the implementation of the Sidra searching system. A baseline configuration of the system was tested and evaluated. Later on, other tests with different parameter settings were run to evaluate the importance that these parameters have on the system performance. The time

Parameter	*Values*
QueryServers	1, 2, 4
Brokers	1, 2, 4
workload	1 - 80
average of terms per query	1.26, 5, 10
frequency of query terms in collection	normal, high
documents returned	10, 100, 1000
collection size/name	3.2M, 44K

Table 3.2: Test parameters

measurements presented, include also the ranking system processing, meaning that they account for documents' search and ranking times.

3.4.1 Test Environment

Test Parameters

Tests were conducted using a variety of parameters with different values shown in Table 3.2. Through their combination, it was evaluated different architecture configurations working with different workloads over different collection scenarios, measuring how the system reacts to changes in these parameters. These tests also allowed me to identify several bottlenecks in the initial designs of Sidra, which were corrected to create its current architecture.

QueryServers and Brokers: by increasing the number of QueryServers and Brokers, the requests are balanced among more of these components. Tests of performance and scalability to the system ranged from 1 QueryServer and 1 Broker, up to 4 QueryServers and 4 Brokers.

Workload: models the number of concurrent Clients, measured in requests per second (req/sec). Workloads varied from 1 req/sec until the maximum tolerated

by the system, 80 req/sec.

Average of terms per query: the 100 most searched queries on tumba! during a one year period, had an average of 1.26 terms per query. The same tests were performed using queries with an average of 5 and 10 terms, built from terms of the same 100 queries.

Frequency of query terms in collection: two sets of terms with different frequencies in the collection were used. The *normal* frequency set, is the same as on the top 100 queries in tumba!. The *high* frequency set, is composed by the top terms with higher frequency in the collection. The set was created with the same 1.26 average terms per query for an exact comparison.

Documents returned: as most search engines, Sidra returns by default 10 results per query. Each result is composed by the title, URL and path to the document. The performance was measured with 10, 100 and 1000 documents on each result set.

Collection size: two collections were used for tests. The first, named 3.2M, is composed by 3,235,140 Web documents of different MIME types from the Portuguese Web. This collection has 78.4 GB of data from which 8.8 GB are text (see [37] for a detailed characterization of this collection). The second, named 44K, is much smaller, composed by 44,271 Web documents from the ul.pt (University of Lisbon) domain.

Testbed

Hardware: 5 computers running RedHat 9 Linux with a 2.4.20 kernel. 4 computers ran QueryServers and Brokers, each with a 2.4 GHz Intel CPU, 1 GB of

RAM and two mirrored disks. Each disk has a rotational speed of 7200rpm, an average seek time of 8.5 ms, and a media transfer rate of 699 Mbits/sec. 1 computer simulated Clients with a dual 2.4 GHz CPU and 4 GB of RAM. All communication between computers took place through a 100 Mbits ethernet.

Methodology

For each test, each Client requested the top 100 queries 3 times. The results presented are the averages of all valid tests. A test was considered valid only when Sidra replied to all requests (no one rejected). This ensures that performance results were observed during a large period and do not correspond to performance peaks.

For a workload w used in tests, each Client sends a different query each $1000/w$ milliseconds, spacing the requests constantly in time. Clients were programmed to not send the same query at the same time, to eliminate some of the cache mechanisms used by the Linux operating system and the BerkeleyDB database (see Chapter 5 for details on the index implementation on top of this database system). BerkeleyDB cache mechanisms improve Sidra's performance because accessed posting lists are cached in memory, and there are overlapping terms among queries [82, 17].

For each query, it was measured the response *time* of each component involved in the process, the time it spent on i/o (denoted *i/o* in the following tables) and in communication with other components (denoted *com*). The remaining time is composed by the CPU time spent by the component and operating system where it runs, and idle times waiting for message replies or scheduling of processes by the operating system (denoted *CPU+idle*). The i/o time was measured as the time spent by all i/o operations used. The communication time was measured as the sum of the sending, receiving and latency times spent on transferring the

messages. To measure the latency time, all computer clocks were synchronized with the NTP (Network Time Protocol) [5]. A timestamp was appended to each message sent. After getting a message, the receiver replied with a timestamp of the reception time. The measured network time is then the difference between timestamps. Computer clocks drift by 0.121 milliseconds with a jitter (dispersion) of 0.257 milliseconds. This drift could be undervalued or overvalued at times. But, since for each message sent another message is received, the differences are balanced. The same technique was used for measuring the communication time between components in the same computer.

3.4.2 Tests

Baseline Tests

The first set of tests formed the baseline for evaluating Sidra. The baseline test system configuration is composed by one QueryServer and one Broker running in the same computer, and several Clients modeled with different workloads. Clients performed the 100 most frequent queries submitted to tumba! under the following conditions:

- average number of terms per query: 1.26
- frequency of query terms in collection: normal
- collection indexed: 3.2M
- documents returned: 10

Response times remained almost constant until the maximum workload supported by Sidra was reached. Times range between 85 and 112 milliseconds for

workload req/sec	QueryServer				Broker				Client
	time	i/o	com	CPU +idle	time	i/o	com	CPU +idle	time
10	79	3	69	7	81	0	71	10	**85**
20	82	3	71	8	84	0	75	9	**89**
30	86	3	75	8	89	0	80	9	**95**
40	92	3	81	8	97	0	90	7	**112**

Table 3.3: Response times in milliseconds for the baseline tests.

a workload of 10 and 40 req/sec, respectively (see Table 3.3). The system performance is limited by the communication time, which reaches up to 93% of the Broker response time, becoming a bottleneck as the workload increases.

Terms per Query

The same tests were conducted for two other sets of 100 queries, with average lengths of 5 and 10 terms. Table 3.4 shows that response times for queries with 5 terms increased 3.4 and 104.6 times, respectively, for a workload of 10 and 20 req/sec compared to the baseline times. After 20 req/sec the system degrades rapidly. The cause of the response time increase in the QueryServer, is that more posting lists need to be read from disk (i/o use), and then decompressed and joined (CPU use); as more data needs to be exchanged between components, the communication also increases. In the Broker, the CPU usage times increase as more data is processed, specially for ranking calculation, where it takes more query terms to compute similarities with the documents.

For the set of queries with 10 terms on average, times increased even more for the same reasons (see Table 3.4).

5 terms per query									
workload req/sec	QueryServer				Broker				Client
	time	i/o	com	CPU +idle	time	i/o	com	CPU +idle	time
10	262	10	65	187	286	0	73	213	**291**
20	2211	11	2090	110	9183	1	2821	6361	**9311**
10 terms per query									
10	480	14	80	386	483	0	83	400	**427**
20	3558	20	2826	712	10002	1	2886	7115	**10014**

Table 3.4: Response times in milliseconds for tests with an average of 5 and 10 terms per query.

workload req/sec	QueryServer				Broker				Client
	time	i/o	com	CPU +idle	time	i/o	com	CPU +idle	time
10	164	22	76	66	169	0	79	90	**177**
20	257	41	91	125	542	0	100	442	**557**

Table 3.5: Response times in milliseconds for tests with high frequency terms in collection.

Frequency of Query Terms in Collection

The same tests were also conducted modifying the query set by choosing terms with the highest term frequency in the collection. These queries represent the worst possible cases in terms of response times. For a workload of 10 and 20 req/sec, Sidra presented response times of 157 and 557 milliseconds, respectively. Response times increased mainly because larger posting lists were read from disk, originating longer decompression and intersection times (see Table 3.5).

Results Returned

In Sidra, the top k ranked documents are returned for every query. The response times of the baseline tests were compared against tests with 100 and 1000 results

100 results returned									
workload req/sec	QueryServer				Broker				Client
	time	i/o	com	CPU +idle	time	i/o	com	CPU +idle	time
10	81	3	69	9	83	2	69	12	**88**
20	84	3	72	9	88	2	75	11	**97**
30	88	3	75	10	92	2	80	10	**102**
40	100	3	86	11	115	2	102	11	**125**
1000 results returned									
10	100	3	71	26	136	25	99	12	**167**
20	362	13	341	8	8824	1287	4992	2545	**9427**

Table 3.6: Response times in milliseconds for tests with 100 and 1000 results returned.

returned, instead of the default 10.

The response times for queries returning 100 results are nearly the same as the ones of the baseline tests (see Table 3.6). For 1000 results returned, the system supports a 20 req/sec workload with a 9427 milliseconds response time. For each query, the Broker had to read documents' metadata from their index, including 1000 titles, URLs and paths for the top 1000 ranked documents matching the query, and then send this metadata to the Client. The results show that this processing originates high i/o times, which cause performance degradation to Sidra. There is also a significant difference in response times between the Broker and the Client, indicating that much time is spent exchanging all this metadata.

Collection Size

It was performed the same tests for a much smaller collection, the 44K collection. As the components have to read and process smaller posting lists, Sidra reaches a superior workload of 50 req/sec, with a smaller response time of 74 milliseconds (see the observed results in Table 3.7).

workload req/sec	QueryServer				Broker				Client
	time	i/o	com	CPU +idle	time	i/o	com	CPU +idle	time
10	65	0	64	1	66	0	65	1	**69**
20	66	0	65	1	68	0	66	3	**70**
30	66	0	65	1	68	0	66	3	**71**
40	67	0	66	1	69	0	67	3	**72**
50	68	0	67	1	70	0	68	3	**74**

Table 3.7: Response times in milliseconds for tests with the 44K collection.

Scalability

Linear scaleup means that, with the double of the hardware a task twice as large can be done in the same time. For a proper analysis if a system scaleup, the number of computers should be multiplied by the same factor as the collection size, and the observed search times should remain constant for a given workload. To evaluate the Sidra's scalability, instead of creating an index of a larger collection and partitioning it by more QueryServers, the index was replicated by all p QueryServers. Then, each one of the QueryServers was responsible to search on a partition with $\frac{1}{p}$ of the lexicon. As posting lists are accessed on disk through a hash table, this corresponds to using one index of a collection p times larger, with a difference: in a larger collection with twice as many documents, the size of the posting lists would tend to be also twice as larger.

1 Broker and 1 QueryServer per computer (1st configuration): The scalability tests started with 1 Broker and 1 QueryServer running in the same computer (this corresponds to the baseline tests). Then, the components were replicated on another computer, creating a configuration with 2 computers, each running 1 QueryServer and 1 Broker. Brokers send requests to both QueryServers in parallel. With this configuration, the system could double the maximum workload,

2 computers (1 Broker and 1 QueryServer per computer)									
workload req/sec	QueryServer				Broker				Client
	time	i/o	com	CPU +idle	time	i/o	com	CPU +idle	time
20	117	3	82	32	118	0	88	30	**122**
40	121	3	86	32	123	0	89	34	**126**
60	127	3	92	32	130	1	95	34	**134**
80	132	3	96	33	136	1	97	38	**141**
4 computers (1 Broker and 1 QueryServer per computer)									
20	209	3	129	77	213	0	140	73	**217**
40	223	3	137	83	235	0	179	56	**239**
60	233	3	143	87	244	1	184	59	**249**
80	286	3	176	107	329	2	268	59	**352**

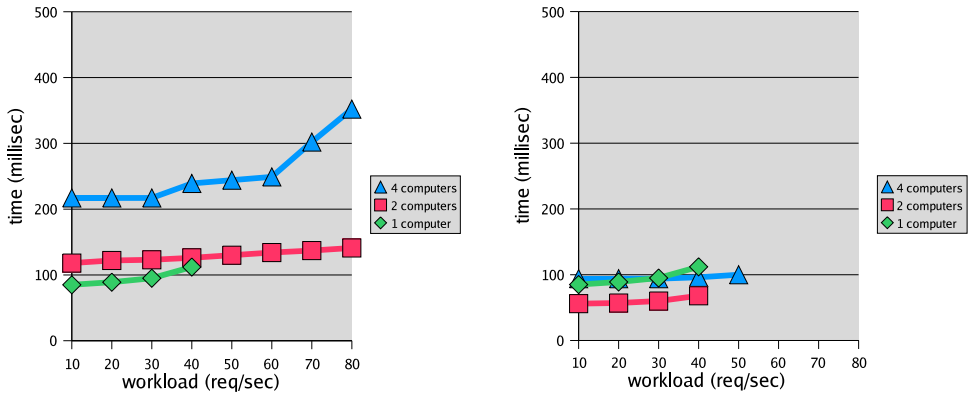
Table 3.8: Response times in milliseconds for tests with 2 and 4 computers (1st configuration).

sustaining 80 req/sec with a response time of 141 milliseconds (see Table 3.8).

After raising the system configuration to 4 QueryServers and 4 Brokers with the addition of two more computers, response times suffer a small increase (see Table 3.8). Figure 3.3(a) shows graphically that there is a sharp raise in time when Sidra goes from 60 to 80 req/sec. Results show that the cause is the increase of the communication time on the Broker.

1 component per computer (2nd configuration): Other configurations are possible when using more than one computer. It was also tested a configuration with a QueryServer and a Broker running in 2 different computers. Response times are shown in Table 3.9. The response times dropped to half with this configuration, but the previous configuration supports a workload twice as large compared to this one (see Figure 3.3(b)).

Another test used 4 computers, 2 running a Broker process each and the other 2 running a QueryServer each. Once again, this configuration presents smaller



(a) 1st Sidra configuration

(b) 2nd Sidra configuration

Figure 3.3: Response times varying the number of computers on the (a) 1st Sidra configuration, and (b) 2nd Sidra configuration.

response times, but supports a lower workload as depicted in Figure 3.3(b). The difference is that, while in the 1st configuration the requests are balanced by more components enabling higher workloads, the 2nd configuration enables smaller response times due to the less concurrency over the computer resources, since it has only one component per computer. According to the results, the 2nd configuration presents the best scalability.

3.5 Results Analysis

Sidra offers very small response times given the processing imposed. For the Web collection tested, these times were under one second even for workloads up to 80 req/sec. Performance peaks reached a workload of 200 req/sec using 4 computers, much more than the necessary for most Web search engines.

In a way or another, all parameters in Table 3.2 influence Sidra's performance, as previously described. However, the measurements have shown that Sidra offers

2 computers (1 component per computer)									
workload req/sec	QueryServer				Broker				Client
	time	i/o	com	CPU +idle	time	i/o	com	CPU +idle	time
10	51	2	41	6	54	0	42	12	**56**
20	51	2	43	5	55	0	44	11	**57**
30	52	2	46	4	58	0	47	11	**60**
40	57	2	52	3	64	0	53	11	**68**
4 computers (1 component per computer)									
10	88	3	59	24	91	0	61	24	**94**
20	88	3	61	24	91	0	62	28	**94**
30	88	3	61	23	91	0	62	28	**94**
40	88	3	63	23	92	0	64	28	**96**
50	91	3	66	23	96	0	67	29	**100**

Table 3.9: Response times in milliseconds for tests with 2 and 4 computers (2nd configuration).

satisfactory results for tumba! and for most large scale Web search engines, even in test cases conceived to deteriorate the performance of the system.

Scalability tests show that response times are small and almost constant as the system scales with different architecture configurations. Despite the good results, several cases of response time degradation were detected:

1. the response time was highly dependent of the speed of the slowest computer in the cluster. It was noticed that one computer used in this tests was significantly slower than the others, delaying response times.

2. some QueryServers received nearly twice as many requests than other QueryServers. To alleviate this, a better distribution of the posting lists based on access statistics would improve response times.

3. there was a high interaction between components, causing permanent waits for replies to continue their processing. This explains why the CPU+idle

time of one component tends to increase, when the CPU+idle time of the other component increases.

4. contrary to the expected, the communication between components through the network (on different computers) presented smaller times than the communication through memory (on the same computer). This result is visible by comparing columns *com* of Tables 3.3 and 3.9. The reason is that the implementation of Sidra creates, to process each new request, a new thread for each component. When the workload is high, each computer has a large number of threads running concurrently, which tends to delay the response of the system. This delay originates in turn a large number of threads running concurrently. This vicious cycle quickly causes a performance degradation on the system. Another problem related to the high number of concurrent threads, is on accepting connections from Clients. Each Broker thread uses a different socket. When the workload is too high, the operating system can not accept connections and free sockets as requested.

The system bottleneck identified as 4 above is responsible for the 1st configuration to present inferior performance than the 2nd configuration. However, the problem is not architectural and could be at least partially solved with a different implementation. A possible solution would be using a limited pool of threads with requests and response queues, instead of creating a new thread per request. Another solution could be adding more computers to the system configuration.

3.5.1 Comparative Results

Table 3.10 presents comparative results, showing Sidra's performance data against similar published results of other distributed IR systems with share-nothing architectures. It is important to underline that all these systems have different scopes,

system	CPU #	collection		index partition	workload req/sec	time sec
		size GB	type			
Sidra 1st conf.	4	313.6	Web	global	80	0.352
Sidra 2nd conf.	4	156.8	Web	global	50	0.1
PADRE97	10	102	text	local	-	78.5
Inquery-based	8	8	text	local	10	1.4
Google	15,000	84,000	Web	local	-	0.55

Table 3.10: Distributed systems to search large scale collections.

differing from Sidra with respect to ranking algorithms, architectures and partitions. Comparisons can only be done coarsely. However, despite the differences, this performance data can still be used as a baseline for relative comparisons.

Hawking developed a system called PADRE97 [43]. It is composed by a Broker that broadcasts requests to QueryServers indexing a subset of the collection documents (local partition).Tests were performed over a 102 GB collection, using 10 computers to search each 10.2 GB of text, with a 200 MHz CPU and 64 MB of memory. It spent 78.5 seconds on average to respond to one query. Linear scaleup was observed on configurations of 9 computers processing a single query.

Cahoon and McKinley developed a simulation model to study distributed share-nothing architectures, using Inquery servers [22]. I call the system Inquery-based. They simulate configurations up to 128 servers, each with an index of a 1 GB text collection. Each server had a 250 MHz CPU with 1 GB of RAM. The architecture is composed by Clients selecting multiple text collections to search simultaneously. In average, they select half of the servers. They send the queries to a Broker, which dispatches them to the respective Inquery servers. Their simulation model, validated with only one Inquery server, predicted that performance exploits parallelism on configurations with up to 32 servers. The best response time for short queries (2 terms on average) was achieved with 8 servers (8 GB). Servers reply on average in 1.4 seconds. For medium (12 terms average) and

long (27 terms average) queries, the best response times were achieved with 128 servers (128 GB), with response times of 119.9 ad 399.9 seconds, respectively. All times were measured for a workload of 10 req/sec.

In the initial prototype of Google, response times over 24 million Web documents lay between 1 and 10 seconds, mostly dominated by disk i/o over NFS (Network File System) [15]. The proponents do not refer the number of computers. In a recent article from Google, they inform that they are using more than 15,000 PCs for Web search [11]. Google front page indicates that the search is done over 4.2 billion documents. Assuming that documents have an average size of 20 KB according studies of the Web [31, 37], currently Google indexes around 84 Terabytes. For comparison purposes, the same 100 queries performed on Sidra's tests were performed on Google. The times on the HTML pages with result sets for these 100 queries averaged to 0.55 seconds.

Table 3.10 shows that Sidra presents better results than the other systems analyzed, with the exception of Google, which operates on a much larger data set. It uses 3,750 times more computers to support searches over a collection 267.8 times larger. Extrapolations on these figures have a small degree of confidence, but assuming that Sidra would scaleup to support a collection of 84 Terabytes as its results show, it would need around 1,000 computers.

3.6 Conclusion

Sidra is a new flexible and distributed query processing system for Web data, supporting different configurations that adapt to diverse performance requirements. The architectural design and the distributed processing was described and validated with tests over the Sidra implementation, using a realistic set of queries extracted from tumba!'s logs. Results show that the system scales and sustains high

performance response times of 100 milliseconds with a workload of 50 req/sec, on a cluster of 4 computers.

Chapter 4

Ranking System

The ranking system orders by relevance the thousands or millions of documents matching a query. It uses several heuristics to compute a relevance value for each document matching a query, and presents the documents sorted by these values. This is a difficult task to accomplish for several reasons. One problem is the size of Web collections that continue to grow and some ranking algorithms do not scale for large dimensions. Most documents aren't relevant to more than a few people, and are intended to be read by humans, not machines. A query is subjective from user to user, meaning that different users expect different results. Web users submit in average only one or two terms per query, sometimes ambiguous, and expect that the ranking system discovers what they have in mind at the moment [78, 46, 79].

Another problem is spam. The inclusion of a Web site among the first results of a Web search engine result set will increase the traffic to that site. Commercial Web sites for instance, have a great benefit when the data of their sites is manipulated to boost their position in the rankings produced by Web search engines. Search engines spam can be performed by manipulation of the contents of documents (e.g. placing usual search terms hidden on special parts of a document),

manipulation of the Web link structure (e.g. placing several references to a site), cloaking (the process of delivering one version of a document to a user and a different version to search engines), or the combination of all the above techniques. For a more detailed description of the Web search engine spam problem see [44].

Ranking algorithms should overcome these problems and be adaptable to the dynamics of the Web. To develop, test and evaluate ranking algorithms, aimed at improving the quality of the results as the Web grows, Sidra provides a software framework, described in Section 4.1. On the internet, users expect no more than a few seconds for a response. Section 4.2 explains how Sidra speeds up the ranking calculation. Section 4.3 summarizes the work on the Sidra ranking system and presents the conclusions derived from its evaluation.

4.1 Ranking Framework

The process of discovering the most relevant results from a large scale collection of documents, give rise to many ranking algorithms and heuristics. This thesis does not intend to present the best ranking algorithms for this purpose. Due to the dynamics of the Web, I believe that the best algorithms of today will not continue to present the same results tomorrow. The past evolution of ranking algorithms in Web search engines shows that. In the first generation of Web search engines, the ranking algorithms used were based on the textual content of the documents. Afterwards, link based algorithms were developed, and currently, ranking algorithms begin to use the semantic data available. All these evolutionary steps were caused by the need to offer better results, when the Web collections grow to different orders of magnitude and their ranking algorithms can no longer satisfy users. This Section describes the Sidra ranking framework, conceived to develop, test and evaluate different ranking algorithms.

4.1.1 Functionality

The ranking function *rank* of a search engine is composed of two sub-functions:

$$rank(d,q) = c * imp(d) + (1-c) * sim(d|q) \tag{4.1}$$

$imp(d)$ weights the *importance* of each document d in the collection independently of a query q. PageRank is an example of an importance function [60, 40]. $sim(d|q)$ weights the *similarity* between a document d and a query q. Algorithm $tf \times idf$ is an example of a similarity function [73]. Both functions can be composed by other functions of the same type. c is a coefficient to balance the weight of both functions.

Sidra has two ranking indexes, one for each type of function. The *sim* functions use the hits index (represented in Figure 3.2) composed by weights computed with the text and layout of the terms occurrences on the documents (e.g. if the term occur on title, URL, anchor, heading, the term frequency, etc). This information is available for each pair <term,document> and is used on algorithms based on textual information. An example of a *sim* function parameterized with this information and used on tumba! with this framework is $termsInTitle(d,q)$, which measures the similarity between the terms of the title of each document d, denoted T, and the query terms from a query q, denoted Q. T has size $|T|$ and Q size $|Q|$.

$$termsInTitle(d,q) = \frac{|T \cap Q|}{max(|T|,|Q|)}$$

This class of functions usually give a similarity value to the documents according to the section where the query terms occur.

The *imp* functions produce importance values for each document, stored on the document-rankingValues index. This index is stored on Brokers and can have several importance measures for each document, as the ones computed with link

analysis algorithms (e.g. [49, 60]), or others as the number of times a document is selected by the users of a system (e.g. as Direct Hit [2] which incorporates the relevancy judgments made by the millions of searchers). In tumba!, this index contains two values produced by two importance measures explained in detail in Section 4.2.3.

Since this is a very dynamic environment, information from other indexes can be used in *rank* function. For instance, Sidra is now using the distance between query terms on documents for ranking tumba!'s results. In this case, an index of positions partitioned as the term-documents index (global partition) is being used too (represented in Figure 3.2). More information about the ranking produced by tumba! is described in [28].

4.1.2 Multi-dimensionality

Multi-dimensional ranking criteria are poorly supported by existent searching and ranking systems. The concern of the majority is centered in providing functionalities to index and search terms in documents. They make available only a part of the information necessary for ranking calculation, mostly statistical data of the documents text and functions over that data.

Sidra is composed by searching and ranking systems working in tandem. Multi-dimensional search indexes enable to restrict and rank results according to the dimensions of search. In Section 3.1 was explained how the multi-dimensionality search is performed. Here, it is described how the results are ranked according to the dimensional context of the query.

Each search dimension could have associated several measures for ranking. The term-documents index has associated the hits index with all the statistical information of the terms occurrences on the documents. The other search dimensions could contain related importance measures associated to each docu-

ment, stored on the document-rankingValues index. For instance, the document-rankingValues index could contain a topic dimension, such as the Topic PageRank [42]. Results would then adapt to the query according to the topic of search. The importance index could have associated other values to each document, as their geographic location. The distance between the user and the document location previously computed, would be used to give a relevance measure for the query. All these ranking values are combined in the *rank* function, which computes a representative value of the relevance of each document for the dimensional context of the query.

4.1.3 Evaluation

To improve the ranking quality of an IR system, it is necessary to evaluate it before and after the changes. This evaluation requires:

1. a search task, delimiting the scope of the quest.
2. a document collection with a set of relevance judgments for each search.
3. metrics to evaluate results.

Search Task

TREC (Text REtrieval Conference) is one of the most important conferences for evaluating IR systems [6]. The evaluation process chosen for the Sidra ranking framework is similar to the one applied on the TREC Web Track competition to evaluate Web IR systems [8]. Web Track has two evaluation tasks. One involves finding entities given their name (named entity task). The other, requires to find relevant sources (documents) matching a topic (topic distillation task).

The named entity task needs to define a representative sample of entities associated to their homepages. This task has the advantage to be simple and it requires

few judgments. On the other hand, the topic distillation task requires the identification of all relevant documents for each topic used. This is a laborious work, specially for collections as large as the ones used with millions of documents. Thus, the named entity task was the only one considered for evaluation on this ranking framework.

Relevance Judgments

TREC experiences showed that evaluation results based on only 5 to 10 topics do not present a high level of confidence. On the other hand, 25 topics are enough to distinguish the ranking quality between IR systems [88, 20]. As the named entity task is more objective than the topic distillation task, it is assumed that no more than 25 entities will be necessary to distinguish the ranking quality of IR systems. For measuring the tumba!'s ranking, a set of 30 homepages of persons and institutions were assembled from the 3.2M collection (detailed in Section 3.4.1), having each entity associated the respective URL and all existing mirrors. The detailed list of entities is described in [77].

Metrics

Many metrics can be used to evaluate the quality of the results returned. The most common are *precision*, which measures how well the system retrieves only relevant documents, and *recall*, which measures how well the system retrieves all the relevant documents [9]. Let $relret_q$ be the number of relevant documents returned, ret_q the number of documents returned, and rel_q the number of relevant documents in the collection, for a query q.

$$precision = \frac{relret_q}{ret_q}$$

year		1998/2000	1998	2002	2002	2003
search engine		Excite	Altavista	Excite	Fast	Tumba!
queries		51,473	1 billion	1,025,910	451,551	356,629
pages viewed	**avg**	2.35	1.39	1.7	2.2	1.46
	1	58%	85.2%	?	?	78.9%
	2	19%	7.5%	?	?	9.6%
	3	9%	3.0%	?	?	4.7%

Table 4.1: Result pages seen by users in Web search engines.

$$recall = \frac{relret_q}{rel_q}$$

However, typical users of Web search engines tend to see only the first pages of results. Jansen et al. analyzed in 1998 (and again in 2000) 51,473 queries of the query log of the Excite search engine [45, 46]. Silverstein et al. analyzed in 1998 approximately 1 billion queries collected over 43 days from the query log of the Altavista search engine [78]. Jansen et al., conducted another study in 2002 where they evaluated and compared the Excite and AllTheWeb search engines [79]. It was also performed an analysis of a subset of the tumba!'s log with 356,629 queries. The collected results are summarized in Table 4.1. They indicate that the majority of users tend to browse only the first two pages of results (top 20 results), sometimes browsing the third result page. Because of that, measures as $precision@10$ (the precision of the first ten results) and $precision@20$ are usually used.

The returned order is also important. The ranking of a system is as better as closer to the first position are returned the homepages of the entities searched. However, $precision@$ measures do not take that into account. Mean reciprocal rank (MRR) is a measuring function that is parameterized with the rank of each query result. The computed result is the reciprocal of the rank at which the first correct response is returned ($\frac{1}{rank}$). The score for a sequence of queries is the

mean of the individual query's reciprocal ranks. According to the results detailed in Table 4.1, it was given a 0 MRR value if a query returns an entity after the 20th rank position. MRR has also the advantage of being related to the average precision measure used extensively in document retrieval [87].

Evaluating tumba!'s ranking

The evaluation proposed was used to evaluate the quality of results returned by tumba!, as well as Google for comparison purposes. Both collections contain all the entities used for evaluation. However, the collection indexed by Google is much larger than the one used on tumba!. On one hand, Google has to search more documents, making the search of good results difficult. On the other hand, there is more information that Google can use to improve ranking, such as a larger Web graph of links between documents. Despite the significant differences, the results may be used to compare, even with some skepticism, the ranking quality of state-of-the-art search engines.

Tumba! got a MRR of 0.883 and Google 0.915, a difference of 0.32. The difference is small. However, for a set of 30 entities, a difference of 0.12 is enough to consider the Google's ranking better in TREC experiments [88]. The ranking of tumba! was improved several times until its current level of quality. New improvements will continue to be studied to eliminate this slight difference between the quality of the two rankings.

This framework was also used on the CLEF Multilingual Question Answering Evaluation [1], to tune Sidra's ranking [25]. Currently, it is being used on the TREC Web Track [8].

4.2 Optimizing Ranking Calculation

The size of collections indexed by some of the major Web search engines reaches today more than 4 billion pages. A search result (list of documents that match a query) can have more than a hundred million pages. If search engines had to calculate the ranking for all these documents, response times would be unacceptable.

Fortunately, most users do not see all the results returned for their searches, but only the first ones. If the documents are sorted in the index by a measure of importance independent of the query, it is possible to restrict the ranking calculation to a subset of the candidate documents containing the documents that users are likely to see. This approach is followed by major search engines to limit response times. Google used to rank a maximum of 40,000 document matches for each query, offering sub-optimal results [15].

In the development of tumba!, it was analyzed techniques to filter irrelevant documents from ranking calculation. A research was performed on how many documents have to be ranked to produce results without loss of quality and how much processing can be saved. Solutions to this problem are necessary now and will be even more in the future, since Web collections tend to grow.

This Section describes the solution that has been devised for this problem.

4.2.1 Preliminaries

The filtering of irrelevant matches before ranking query results has been widely studied for decades in many querying scenarios. Initially, ranking algorithms were based on statistics about the textual content of the documents and collections [73]. The first developed pruning algorithms were based on these statistics. As other data began to be used in ranking calculation, specially Web linkage, pruning algorithms also evolved to account for this new information.

Independently of the information used, pruning algorithms may be classified as either safe or unsafe. Safe algorithms reduce computation without affecting results [19, 91, 63, 30, 58, 39, 18]. Unsafe algorithms trade quality of results for speed, by relaxing mathematical guarantees of the results' correctness [56, 26, 51]. Both classes of algorithms operate in general over inverted indexes.

4.2.2 Reducing Ranking Calculation

Search engines results are produced in two steps. First, they match the documents that satisfy a query q. These are called *query matches for* q, denoted as $QMatches_q$. Then, search engines rank each document d in $QMatches_q$, applying a ranking function *rank* to produce a list of the query results for the user, denoted as $\left[QMatches_q\right]_{rank}$.

Evidence shows that on most queries, users only see the top k documents of $\left[QMatches_q\right]_{rank}$, represented as $\left[QMatches_q\right]^k_{rank}$. Therefore, it isn't necessary to compute the ranking score for all the documents in $QMatches_q$. Equation 4.1 shows that a ranking function *rank* of a search engine is composed of two subfunctions: a $imp(d)$ function that weights the *importance* of each document d in the collection independently of the query; and a $sim(d|q)$ function that weights the *similarity* between a document d and a query q.

The technique developed to reduce the ranking calculation, is based on filtering those documents d whose importance $imp(d)$ is not large enough to rank them among the most relevant, independently of the query q. First, all $imp(d)$ scores are computed for each document d in the collection, and the posting lists sorted by these scores. As the function *imp* is independent of the query, this processing can and should be performed offline.

When a query is processed, the top n documents in $\left[QMatches_q\right]_{imp}$ are selected, denoted as $\left[QMatches_q\right]^n_{imp}$. The value of n should be sufficiently large to

contain all the documents that would be seen by the user if all matching documents were ranked, but as small as possible to reduce maximally the number of documents to compute the ranking. If n is large enough for the subset $\left[QMatches_q\right]_{imp}^{n}$ to contain all documents of subset $\left[QMatches_q\right]_{rank}^{k}$, after applying the function *rank* to both subsets, the rankings produced will be identical for the top k documents. So, it is necessary to find a n such that:

$$\left[\left[QMatches_q\right]_{imp}^{n}\right]_{rank}^{k} = \left[QMatches_q\right]_{rank}^{k}$$

After finding n, it is only required to compute online the similarity between query q and the top n documents that match q, to obtain the same top k ranked documents for q. To find n, is necessary a $\widetilde{n}$ function that computes for any query q the value of n .

Dissected the problem, the solution needs to find the best values for the variables involved in the problem of reducing ranking calculation. That is, for a set of queries Q with size $|Q|$, find the highest reduction in ranking calculation evaluated with the *reduction* function:

$$reduction(k,\widetilde{n}) = 1 - \frac{\sum_{i=1}^{|Q|} \widetilde{n}(|QMatches_i|,k)}{\sum_{i=1}^{|Q|} |QMatches_i|}$$

4.2.3 Searching for a Good Solution

As the *imp* and *sim* functions, and the constant c in the rank function are arbitrary, it is impossible to find a universal solution to the problem independent of these functions. This sub-section presents ranges and alternatives for the various parameters of the *reduction* function, based on the data collected from the tumba! search engine.

Choosing k

In most cases, users only see a small fraction of all the results that match a query. In a typical interaction between a user and a search engine, users make queries to search engines and receive a list of linked results, normally 10. These results are ranked by their relevance to the query.

Table 4.1 indicates that the majority of users tend to see only the first two pages of results (top 20 results), and sometimes the third results page. k should have a value high enough to cover the results that most users usually see, so it was evaluated the *reduction* for $k = 10$, 20 and 30. Considering a value of k higher than 30 (3 results pages with 10 results each), would produce a negligible difference in observed results for the vast majority of queries.

Choosing Ranking Functions

imp **functions.** To evaluate the effect of *imp* in the reduction of calculations, it was chosen 2 functions. One is a variation of PageRank called extPageRank. PageRank computes an importance value for each page using the Web graph with an equal and full flow in all the links. extPageRank assigns a 10% flow to internal links (between the same site), since most of them are navigational and do not correspond to an importance given by independent authors. This percentage was tuned empirically.

The distribution of extPageRank follows a power law distribution (see Figure 4.1). This is similar to the PageRank distribution [61, 41, 85]. The higher extPageRank values are very sparse and differences among them do no reflect the disparity of importance. The extPageRank values were segmented and assigned a weight to each of these segments (see Table 4.2). This seems to be what Google does, based on the observed normalized scores between 0 and 10 shown on the Google toolbar [85].

imp functions		weight
extPageRank	URL weighting	
[0,0.001[	file	0
[0.001,0.01[	path	0.25
[0.01,0.1[	subroot	0.5
[0.1,1]	root	1

Table 4.2: Weights given to each class of *imp* functions.

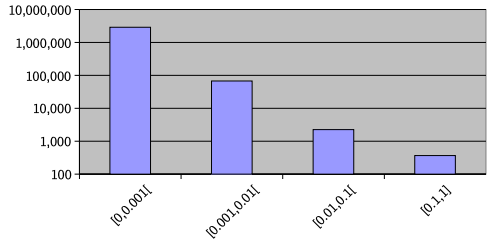


Figure 4.1: extPageRank distribution.

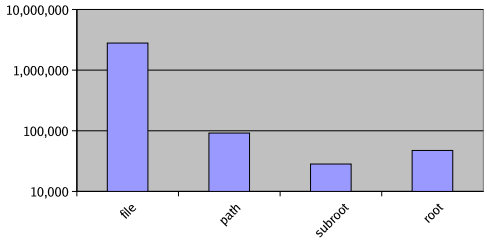


Figure 4.2: URL weight distribution.

The second *imp* function is computed as the URL weighting algorithm that achieved the best results on the TREC-2001 home page finding task [89, 50]. It is based on the observation that documents at the root URL of a specific site are often entry pages. As we descend deeper in the site's directory tree, the probability of being an entry page seems to be inversely proportional. In this algorithm, the URLs are normalized and then divided in four types:

root: a domain name, e.g. tumba.pt

subroot: a domain name followed by a single directory, e.g. tumba.pt/pt/

path: a domain name followed by more than one directory, e.g. tumba.pt/pt/tek/

file: anything ending in a filename, e.g tumba.pt/pt/about.html

The URL class distribution (see Figure 4.2) also has similarities with other works [89, 50, 86]. The difference is that the crawl of the Portuguese Web has

more pages belonging to the root class than to the subroot class. I conjecture that this fact results from having used as seeds, a list of registered Web domains whose Web sites mostly have only a homepage. Each class was weighted as shown in Table 4.2.

The two distributions have a correlation of 0.477.

***sim* functions.** The *sim* function adds similarities between query terms and documents, using different weights for special sections of the documents where the terms occur (e.g. in URLs, titles, anchors). The scores produced by the *sim* function are normalized between 0 and 1.

coefficient c. The coefficient c balances the relative weight of the *imp* and *sim* functions. It was set in this study to values of 0.5, 0.66 and 0.8, corresponding to weight ratios of $\frac{c}{1-c} = 1, 2$ and 4.

Determining $\widetilde{n}$

The function $\widetilde{n}$ represents the distribution of n for a query set Q. As this distribution should contain n values as small as possible to reduce ranking calculation, pruning algorithms were used to compute these n values over the inverted index sorted by a *imp* function.

In the optimization of the tumba! ranking algorithm it was studied two representative algorithms, one safe and another unsafe. As safe algorithm, it was used TA-Adapt on posting lists pre-sorted by *imp* [18]. TA-Adapt guarantees that the results of the top k documents are produced as if the ranking would be computed for all the documents.

TA-Adapt is a variant of the TA algorithm [30, 58, 39], which uses only one sorted index to access values sequentially and the other indexes randomly. Let

the threshold t be the lowest ranking between all the documents in the set of k candidates, and $U(d)$ the upper limit that a ranking value of a document d can have. $U(d)$ is computed with the known ranking value from the sorted list, and the remaining ranking values are set to the maximum value they can have. TA-Adapt first gets the top k documents from the sorted inverted list, the candidates set, and computes their ranking probing the other ranking values from other sources. Then, TA-Adapt repeatedly accesses the next document d from the sorted inverted list until $U(d) < t$. When this happens, it stops and returns the top k documents from the candidates. Until then, if $U(d) \geq t$, the ranking of d is computed with all the probed ranking values and compared against t. If the ranking value is superior to t, the document is added to candidates and t is recomputed, or ignored otherwise. The $\widetilde{n}$ function representing the distribution of the TA-Adapt algorithm is denoted as *safe/TA-Adapt*.

As unsafe algorithm, it was studied one based on the analysis of tumba!'s statistical data on queries and documents, denoted Stat. This algorithm first computes, for each query, the ranking for all matching documents and then gets the top k results. Then, it fetches the documents by the *imp* sorted order of the inverted index posting lists, until collecting the same k documents. The number of fetched documents is the n value. Computing a linear regression function using these n values, results in a function $\widetilde{n}$ that statistically predicts n for any query. This function is denoted as *unsafe/Stat* $\simeq n$. However, as this function does not completely cover all n documents, it was also computed a linear function $\widetilde{n}$ that covers all n documents and offers the highest possible reduction. Its coefficient was computed as the highest value of $\frac{n}{|QMatches_q|}$ for all queries. This function is denoted as *unsafe/Stat* $\geq n$.

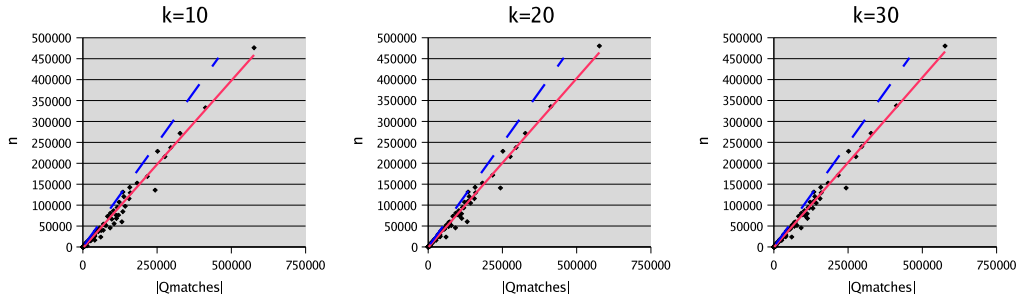


Figure 4.3: $\widetilde{n}$ as a function of n using Stat with the collection index sorted by extPageRank.

4.2.4 Results

The reductions that could be achieved were observed on the 3.2M Web collection. For the 100 most frequent queries submitted to tumba! during a one year period, it was analyzed the distribution of n for the *unsafe/Stat* $\simeq n$, *unsafe/Stat* $\geq n$ and *safe/TA-Adapt* functions, with the extPageRank and the URL weighting algorithms as *imp* functions. Results were produced for the three k values considered: 10, 20 and 30, always using a coefficient c equal to 0.5. At the end of this subsection, it will be shown how reductions change as c varies.

Unsafe/Stat $\simeq n$ function

***imp* as extPageRank.** Using the Stat algorithm over the index sorted by extPageRank, it was computed a n value for each of the queries. As the graphics in Figure 4.3 show, n increases almost linearly with $|QMatches_q|$. By applying linear regression to n values, a function $\widetilde{n}$ which predicts n for each query was defined. It is parameterized with $|QMatches_q|$ and k. This function is depicted as the continuous line in the graphics of Figure 4.3. For the three different values of k considered, $\widetilde{n}$ was computed. As k increases, n increases a bit. This increase is expected, since more documents of the subset $\left[QMatches_q\right]_{imp}^{n}$ become necessary

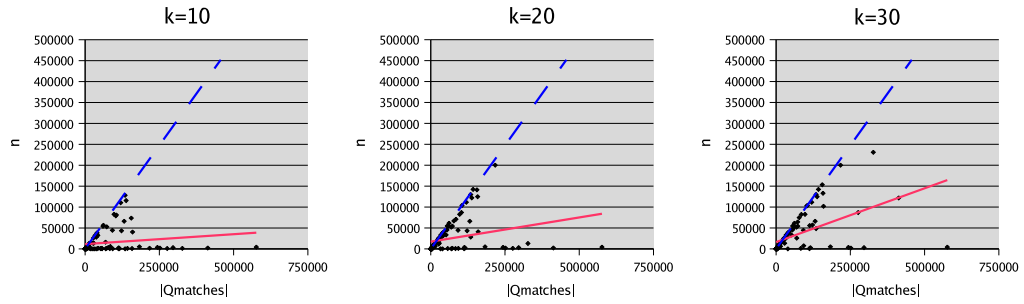


Figure 4.4: $\widetilde{n}$ as a function of n using Stat with the collection index sorted by URL weight.

to ensure it contains the additional documents in $\left[QMatches_q\right]_{rank}^{k}$.

All graphics show that n is smaller than $|QMatches_q|$. Therefore, it is possible to achieve a reduction in ranking calculation. Figure 4.7 plots the reductions achieved in ranking calculation. The number of documents ranked was reduced by 24.13% for a k of 10, 21.35% for a k of 20, and 20.72% for a k of 30.

imp **as URL weighting.** Doing the same tests with Stat over the Web collection index sorted by URL weight, it was computed the distribution of n presented in Figure 4.4 for the different k values. Unlike extPageRank, this algorithm does not present a linear distribution of n and consequently, the function produced using the linear regression of these points can not predict much.

Unsafe/Stat $\geq n$ function

Independently of the *imp* function used, $\frac{n}{|QMatches_q|} \simeq 1$ for all n values, so almost all the documents must be ranked to get optimal results (depicted as the dashed line in the graphics of Figure 4.3 and Figure 4.4).

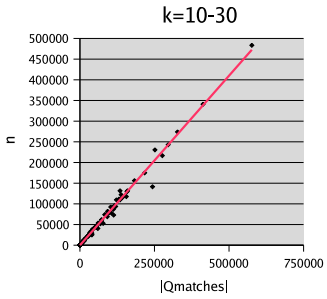


Figure 4.5: $\widetilde{n}$ as a function of n using TA-Adapt with the collection index sorted by extPageRank.

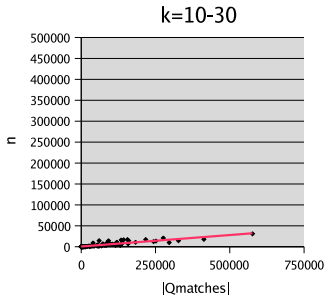


Figure 4.6: $\widetilde{n}$ as a function of n using TA-Adapt with the collection index sorted by URL weight.

Safe/TA-Adapt function

***imp* as extPageRank.** Using the TA-Adapt algorithm over the index sorted by the extPageRank algorithm, a slightly lower reduction than the one achieved with Stat was achieved (see Figure 4.5). The reduction is always 18.33% for the three k values, as depicted in Figure 4.7. The justification is that for each query, the top n documents ranked with extPageRank have an identical *imp* weight.

***imp* as URL weighting.** It was achieved very good results using the TA-adapt algorithm over the index sorted by URL weight (see Figure 4.6). The top 10 to 30 results got a 93.71% reduction (see Figure 4.7).

varying coefficient c

In previous results, the coefficient c always had the value of 0.5 ($\frac{c}{1-c} = 1$) to balance the contributions of the *imp* and *sim* functions to the final rank. To measure the impact of this parameter in observed results, the same tests were conducted varying the coefficient c to 0.66 and 0.8, corresponding to $\frac{c}{1-c} = 2$ and 4. Results are presented in Figure 4.8 for a $k = 10$.

The reduction increased for all algorithms. As the weight of *imp* increases, the

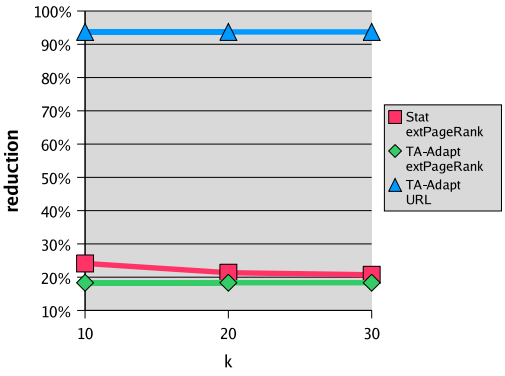


Figure 4.7: Reductions observed using several algorithm combinations in the *rank* function.

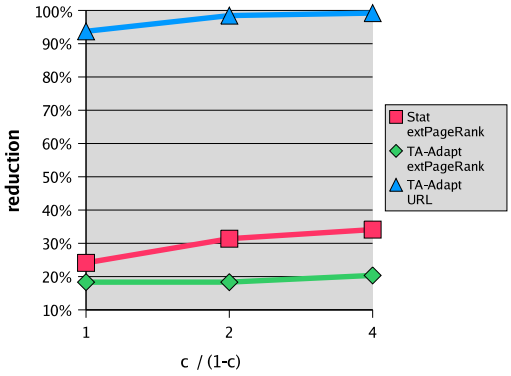


Figure 4.8: Reductions observed using several values for the coefficient c in the *rank* function.

similarity between the rankings of the *imp* and *rank* functions also increases. For $\frac{c}{1-c} = 4$, the algorithms Stat using extPageRank, TA-Adapt using extPageRank and TA-Adapt using URL weighting, achieved a reduction of ranking calculation of 34.13%, 20.35% and 99.18%, respectively.

4.2.5 Results Analysis

The *unsafe/Stat* $\simeq n$ function can only be applied to reduce ranking calculation, when the index is sorted by the extPageRank algorithm. The distribution of n is close to its linear regression, so it can statistically predict a value n for all queries. It reduced ranking calculation by 24.13% when only the top 10 results needed to be exact. When the index is sorted by the URL weighting algorithm, n becomes very difficult to predict.

The *unsafe/Stat* $\geq n$ function, which covers statistically all n values, must rank almost all the documents to get optimal results. This makes this function a bad choice, independently of the *imp* function used.

Using the *safe/TA-Adapt* function, it is guaranteed that $\widetilde{n}$ covers mathematically all the n documents and consequently all the top k results are identical as

the ones produced if the ranking was calculated for all the matching documents. Using it with the extPageRank algorithm to sort the index, the ranking calculation was reduced by 18.33%. With the URL weighting algorithm, the reduction raised to 93.71%. This large increase results from differences in the distributions of the extPageRank and URL weighting algorithms (see Figure 4.1 and Figure 4.2). extPageRank has much less documents in the classes with higher weight than the URL weighting algorithm. Consequently, algorithms with a power law distribution, such as extPageRank, tend to return most of the documents with the same *imp* value, the one assigned to lower classes. It become then difficult to distinguish the more important documents from the less important. Thus, safe algorithms must evaluate many documents before finding a document which mathematically guarantees that it is safe to prune the rest of the documents. Stat presents better results for this class of algorithms. On the other hand, algorithms with a more uniform distribution, such as the URL weighting algorithm, have better results when combined with safe algorithms. This uniform distribution spreads much more documents in the higher classes, which tend to fill the first positions of the posting lists, enabling to quickly select the most important documents for ranking calculation.

There is another important perspective to compare these algorithms. Web search engines of today index up to billions of documents and their indexes need to be partitioned by clusters of machines. The partition can be local or global. With the local partition, all QueryServers rank their subset of documents and the top k are then merged by a Broker. With the global partition, the one used by Sidra, sets of documents are repeatedly requested from QueryServers with query terms indexed, and intersect as necessary until having the top k documents ranked. The algorithms analyzed not only reduce computation, but also enable a reduction of the number of postings fetched on disk. They also enable to reduce the num-

ber of messages, requesting more documents when using global partition. Safe algorithms need to request sets of documents as necessary, until having the n documents that guaranty the top k documents ranked. Recall that safe algorithms only know n after evaluating n documents. The statistical algorithm used, Stat, can predict n without previously requesting any document. Therefore, combining safe algorithms with Stat, Sidra gets the best of two worlds:

1. safe algorithms warrant that results are exactly the same as ranking all matching documents.

2. safe algorithms also offer the best reduction, because the index is sorted by a ranking algorithm with uniform distribution.

3. Stat can compute a statistical function $\widetilde{n}$ for a safe algorithm (e.g. in Figures 4.5 and 4.6), to predict n apriori and minimize the number of requests.

4.2.6 Number of Query Matches

The optimization of ranking calculation creates a new problem. Since the posting lists are only partially intersect, Sidra does not know the total number of documents matching a query q. This number is a useful indicator for users, typically shown by Web search engines. Sidra estimates the number of matches extrapolating the matches until n.

Let $nSidsTotal_i$, $nSidsRead_i$, $nSidsJoined$ be, respectively, the total number of sids, the number of sids read and the number of sids joined until n, for a query term i. The number of query matches is computed as:

$$|QMatches_q| = min(\frac{nSidsTotal_i}{\frac{nSidsRead_i}{nSidsJoined}}) \qquad \text{for all } i \in q$$

The result of the function tends to produce an accurate number of query matches from empirical evaluations.

4.3 Conclusion

As Web users and Web collections tend to grow, it becomes more difficult for Web search engines to offer fast results with good quality. Sidra offers a framework to develop, test and evaluate ranking algorithms, designed to improve and adapt the quality of results to the dynamics of the Web. It follows an evaluation methodology similar to the one used on the TREC Web Track.

An evaluation of the tumba!'s ranking quality supported by this framework was performed, and results showed that tumba! has a ranking quality close to Google in named entity recognition tasks. New improvements are being studied with the help of this framework to eliminate this small difference.

The Sidra framework also provides support for multi-dimensional ranking of results over available search dimensions. Ranking algorithms could be enrich with semantic information to adapt their results to query contexts. For instance, the results for a user in Lisbon searching for a bar should not be the same as one searching from London.

Another problem is to compute relevance values for all the documents on large Web collections. This requires much more time than the few seconds that users tolerate. As users of Web search engines tend to see on average only the first two pages of results, it is unnecessary to calculate the ranking for all the documents that match a query. Only a subset of the candidate documents that contains the documents that users usually see need to be ranked. The evaluation of combinations of algorithms to filter documents considered irrelevant for ranking calculation, lead to the conclusion that the TA-Adapt algorithm over a Web collection

index sorted by URL weights, reduces by 93% the number of documents to rank with no difference in the results seen by end users.

It was also developed a statistical algorithm which can accurately predict the number of postings that have to be fetched from disk in a single i/o transfer, and requested from remote QueryServers, significantly reducing the number of exchanged messages during query processing.

Chapter 5

Indexing System

IR systems use word indexes to speed up the search of relevant documents in a collection. Building these indexes for small collections of documents is a well studied and relatively easy to accomplish process. However, building a distributed index for large Web collections is a much more complex task. The Web continues to grow and modern Web search engines only index a small part of it. The deep Web, the part of the Web stored in online databases, is estimated to be 500 times larger than the known Web and still remains to be indexed [12].

Three factors make the development techniques for building large scale indexes a challenge. Fast indexing is a desirable feature, because Web search engines usually have time restrictions to update their indexes with fresher information. The system must be prepared to scale for the fast and unpredictable growth rate of Web collections, with the addition of inexpensive PCs. Specialized techniques are demanded for efficient indexing, overcoming the limitations imposed by the lack of memory and storage for collections of this magnitude.

This Chapter details the Sidra indexing architecture and the algorithms implemented for the parallel and distributed creation of large scale Web indexes. In Section 5.1, some background is given on how a centralized indexing algorithm

is usually implemented. In Section 5.2, the architecture and the algorithms implemented in Sidra are described, while Section 5.3 presents the results achieved during the indexing of the 3.2M collection delimited as the Portuguese Web. Section 5.4 explains why Sidra adopted index reconstruction instead of supporting partial updates to indexes, and Section 5.5 presents the conclusions.

5.1 Centralized Indexing Algorithm

The indexes of large collections of documents can not be built in memory in a single step, given their size. First, the data of the index must be broken in smaller parts than the memory available. Then, each part is processed separately. An extensive analysis performed by Moffat, compares ten algorithms to build inverted indexes [54]. Some are unfeasible due to the large quantity of memory or storage needed, others because of the time spent in processing. As most of the time is spent in random disk seeks, algorithms with sequential access to files are the most efficient. Indexes are typically created in four phases:

1. Read and parse the documents. Each extracted token (a word separated by spaces), called a term, is stored on a temporary file with associated information. Each term is associated on a triplet <*term,docId,pos*>, with the identifier of the document, *docId*, and the position of the term in the document, *pos*. Other possible information may be added to each triplet, such as the font characteristics. Each of the triplets on the resulting file is called a *hit*.

2. Generate runs. As the hits of a large scale collection do not all fit in main memory, an external sort algorithm is used to sort them in batches, seeking to minimize the cost of disk accesses. Hits are divided in blocks of approximately the size of the available memory. These blocks are then sorted by

term, id and position, in this order, using an in-memory sorting algorithm. Each of these sorted blocks stored on disk is called a *run*. The result is a set of runs.

3. Merge all runs (the next phase of the external sort). Runs are merged in pairs until only a single run with all the sorted hits remains. This can be accomplished by any external sorting program, such as the UNIX sort.

4. Generate an inverted file. In this indexed file, terms are the keys and the value of each key is the list of hits containing the term. To create it, hits are sequentially read from the run produced in the previous phase. Usually, the inverted file is compressed to reduce storage size and i/o transfer times.

5.2 Sidra's Distributed Indexing Algorithm

Unlike centralized indexes discussed above, Sidra can support multiple indexes, organized and partitioned by different criteria. Each index is used to search on a dimension of the data. Each partition indexes a subset of the data of a dimension.

To improve scalability of the creation of indexes of large Web collections, it was developed a parallel distributed variation of the typical centralized indexing algorithm presented in the previous Section. This algorithm was designed to work over an environment of computer clusters, sharing nothing but the underlying network. Figure 5.1 depicts the architecture of the Sidra index generation system. The documents to index must have been previously crawled from the Web and stored in a Web repository. Indexing starts once the crawl is archived. The remainder of this section describes each of the phases of the distributed indexing algorithm.

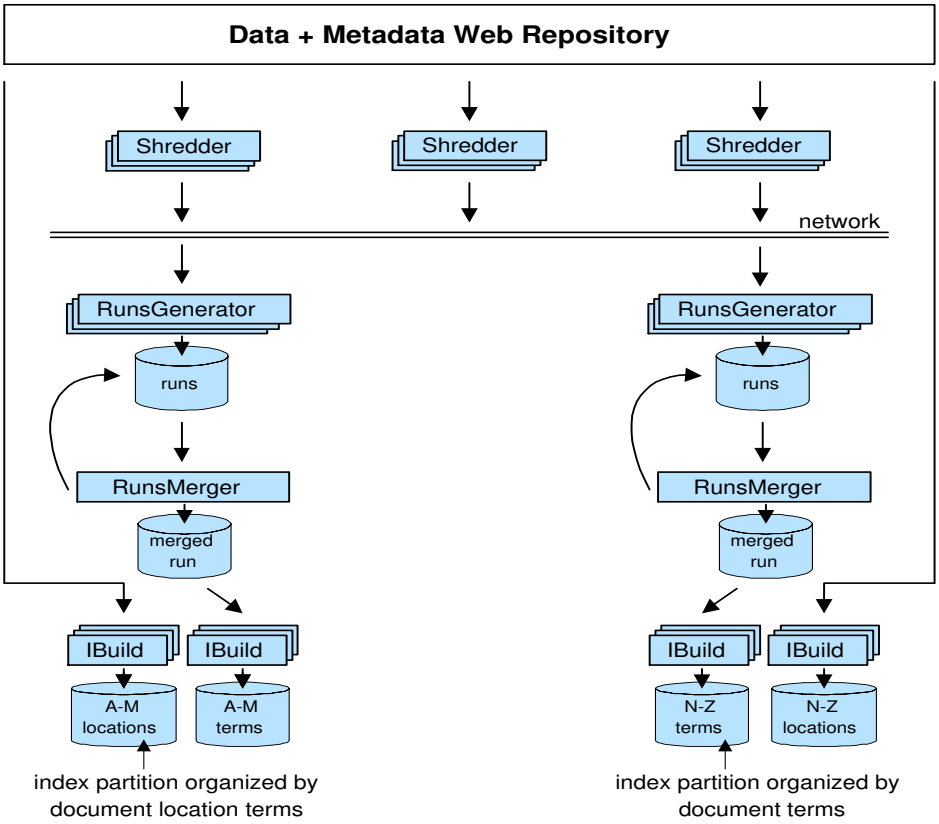


Figure 5.1: Sidra's distributed architecture to create indexes.

5.2.1 Generating Runs

A multi-threaded program, the Shredder, parses documents and associated metadata contained in the Web repository. In general, several Shredders run from a set of computers. The number and capacity of computers is defined based on the time that can be spent on processing the data to index. Shredders extract hits, identical to those of the first phase of the centralized indexing algorithm.

Parsing the wide heterogeneity of Web pages is a complex task. The parser must be robust and tolerant against errors in Web pages. Sidra incorporates a Web data parser developed for tumba!, called WebCAT, which handles a wide variety of Web page formats, as HTML, MS Office formats, PDF (Portable Document

Format), XML and more [52].

In Sidra, the parsed hits are not stored locally on disk. Instead, hits are directly sent to remote processes called RunsGenerators, that collect them to create index partitions. This way, it is eliminated all i/o operations that would be required to write and then read the runs locally. Each RunsGenerator receives the hits assigned to its partition from all the Shredders, sorts them with the quicksort internal sorting algorithm, and then saves them into a run. To avoid network bottlenecks, Shredders send messages containing a small number of hits as they become available. However, this number of hits should be large enough to avoid penalties due to send many messages. This technique is derived from the RR algorithm, proposed by Ribeiro-Neto et al. [69].

Sidra combines the RR technique with a technique presented by Melnik et al., based on software pipelines [53]. The creation of runs is partitioned in three distinct phases: loading, processing and flushing. During the loading phase, a number of pages are read from the network to memory. In the processing phase, hits are parsed and sorted using mostly CPU. In the flushing phase, the runs are stored on disk. These three phases are iteratively executed in pipeline until no more pages remain to process. As each phase uses different resources, good concurrency is achieved through the parallelization of these three phases. There is optimal concurrency when all the resources are used simultaneously.

5.2.2 Merging Runs

At the end of the first phase, each RunsGenerator has received the hits necessary to create an index partition. The hits are organized and sorted in a set of files, the runs. On the second phase, the RunsMerger merges sets of runs iteratively, until a single hits list file remains. It was implemented several merge algorithms to identify which one offers the best response times. A multiway merge algorithm was

chosen with a replacement selection technique using a heap data structure [74]. The algorithm also makes use of a double buffering technique, used in database management systems to maintain the CPU busy during i/o requests [65]. Double buffering requires, for each buffer of a run, another buffer of the same size. When no more hits remain to be read from one buffer, they are read from the second buffer. At the same time, another thread fills up the empty buffer in parallel with hits read from the run file. This way, CPU idle times originated from i/o operations are eliminated. The same technique is applied to the output.

5.2.3 Building Inverted Files

At the start of this phase, all the hits of each partition are sorted in one single run. The creation of an inverted file is now a simple process. An index building program (IBuild) reads the hits sequentially and creates a posting list for each distinct term.

Sidra uses different indexes to search documents by different dimensions. Other IBuild applications read in parallel the metadata associated to the documents from the Web repository, and create partitions of index files corresponding to other index dimensions. Figure 5.1 shows two possible partitioned indexes. One is the usual term-documents index. The other is a geographic index indicating the documents associated to terms representing locations composed by metadata from the Web repository.

In Sidra, posting lists are compressed with the Binary Interpolative Coding, a compression algorithm of integer posting lists, to minimize storage requirements and i/o latency [55]. This compression algorithm was chosen because it offers the best compression ratio and is one of the fastest. Chapter 4 showed that to speed up searches, Sidra only rank the subset of the most important matching documents to any given query. Therefore, it would be a waste of time decompressing whole

posting lists to only extract a subset of the postings. Posting lists are compressed in segments of a fixed number of sorted sids. In query processing, segments of posting lists are sequentially decompressed as needed until the Broker have the desired subset of matching documents.

Some indexing systems use DBMSs to store inverted files [17, 32, 38]. This allows a faster development, since transaction support is already available and there are tools to visualize and easily update the data. However, DBMSs do not enable a fine grained control over the data structures and core parts of the processing, necessary for a high performance system. Sidra, uses BerkeleyDB hash tables to manage and store indexes. BerkeleyDB is an open source embedded database library that provides scalable, high-performance, and concurrent data management services to applications through an API [59]. BerkeleyDB provides a panoply of functionalities that reduced the development time of Sidra while enabling the optimization of the core parts of its processing and storage modules. Sidra uses hash tables because is the fastest way to access the posting lists. There is a very small probability of collision with the implemented FNV hashing function (see `http://www.isthe.com/chongo/tech/comp/fnv/`), tolerable for a system of this nature.

5.2.4 Fault Tolerance

As index generation is a very time consuming process, it is not viable to reconstruct the index from the beginning in case of failure of any of the Sidra's components. It is necessary to have mechanisms to resume the construction of the indexes from the point where it stops. The first phase of the index construction (generation of runs) is the most time consuming. Therefore, fault tolerance mechanisms was implemented for that phase. The other phases are relatively fast, so the construction from the beginning does not present a real problem.

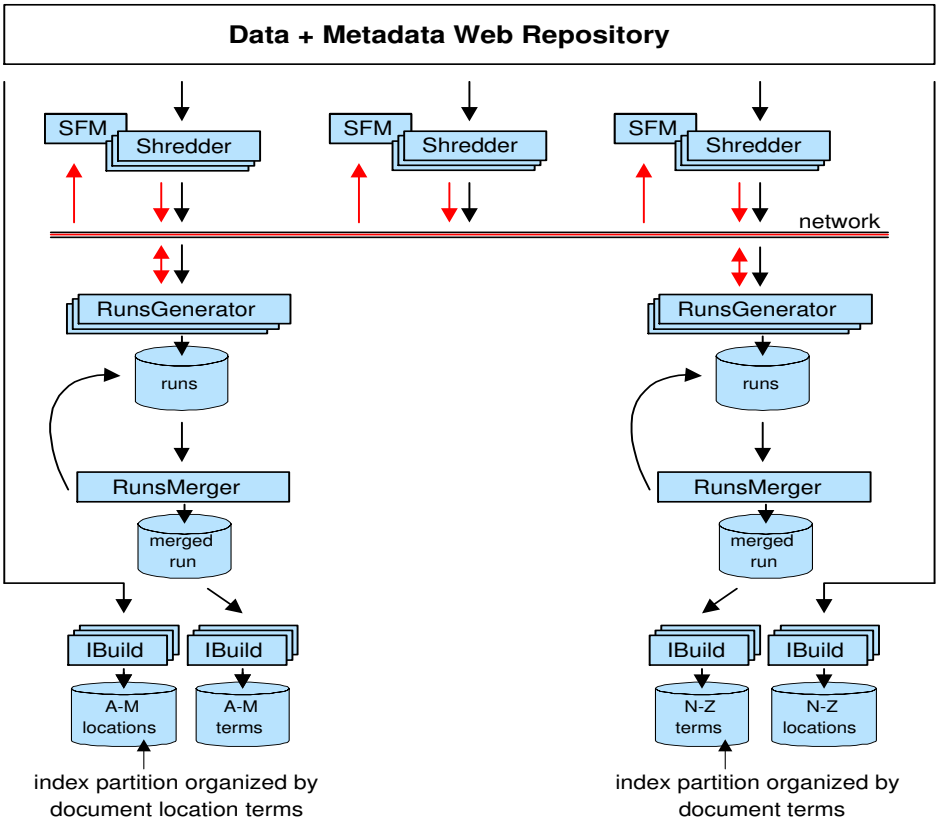


Figure 5.2: Sidra's distributed architecture to create indexes with fault tolerance.

In the first phase, each computer hosting a Shredder, also contains a Shredder Fault Manager (SFM) component (see Figure 5.2). Fault tolerance is achieved as follows:

- when a Shredder ends parsing a set of documents and associated information, it notifies all RunsGenerator components.
- after a RunsGenerator stores a run persistently on disk, it notifies the SFM associated to the Shredders that all hits from the run are stored persistently on disk.
- after a SFM is notified by all RunsGenarator components, it writes on disk

the documents that have all the information stored persistently.

- when the Shredder restarts, the SFM informs which documents have already been processed after read this information locally.

A fault can occur in several points. It may occur before the Shredder notifies the RunsGenerator components, before a RunsGenerator notifies the SFM, or before the SFM writes on disk the documents processed that have all the information stored persistently. In all these cases, a Shredder will process again some documents, and send duplicated information to the RunsGenerator component. To ensure consistency, the RunsMerger eliminates duplicated hits.

The Fault tolerance mechanism design is too complex. It would be simpler and faster if the communication was one-directional from Shredders to RunsGenerators. The Shredders only have to send hits to the RunsGenerators and notify them when the document is already processed. When the Shredder restarts, the Shredder would read information from remote logs available on the RunsGenerators computers.

The Sidra system also includes one software watchdog for each of its main components, that verifies from time to time if the component is alive. Each component has a special thread that writes the system time periodically to disk. The watchdog verifies within the same periodicity if the time has changed. After 3 attempts reading the same time, all components are reinitialized using the fault recovery mechanisms.

5.3 Results

This section presents the performance and scalability results obtained with the existing implementation of Sidra indexing the 3.2M collection of Web documents.

5.3.1 Testbed

Tests were performed on the same cluster of 4 computers used for the searching system tests (see Section 3.4.1).

The Collection used is the 3.2M collection delimited as the Portuguese Web and used on the searching system tests. It is composed by 3,235,140 Web documents, totaling 78.4 GB of data from which 8.8 GB are text [37].

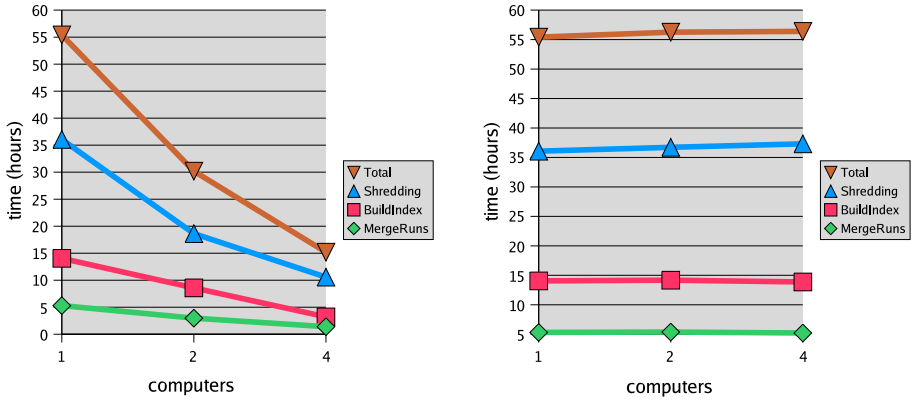
During the first phase of the index creation test, Sidra's configuration had one Shredder and one RunsGenerator component per computer. The next phases ran the remaining components independently in each of the available computers.

5.3.2 Tests and Analysis

Sidra indexed the 3.2M Web collection in 55.41 hours using a single computer. 20.2 hours of this time were spent retrieving and decompressing (with zlib) the documents from the Web repository.

The same collection was also indexed with 2 and 4 computers. Run time decreased to 30.19 and 15.19 hours, respectively (see Figure 5.3(a)). These times show that Sidra exhibits nearly linear speedup in all the processing phases. This means that with p times more computers, the indexing time drops to $\frac{1}{p}$ of the total, which is essential to produce fresher indexes.

Sidra also indexed the same collection replicated in the computers, simulating a collection 2 and 4 times larger. Times are almost constant, varying between 55.41 hours, when the collection is indexed with 1 computer, and 56.38 hours to index a collection 4 times larger with 4 times more computers (see Figure 5.3(b)). Sidra shows basically a linear scaleup. This means that, with p times more computers, Sidra can index a collection p times larger in the same time.



(a) speedup (b) scaleup

Figure 5.3: Times to index the 3.2M Web collection varying (a) the number of computers, and (b) the number of computers and the collection size the same order of times.

5.3.3 Comparative Results

Table 5.1 presents comparative results, showing Sidra's performance data against similar published results. All these systems have different scopes and aims. Some indexed only textual data, while others parse Web documents. Some decompress documents before indexing, while others do not. The measures were obtained in different hardware configurations. However, despite the differences, this performance data can still be used as a baseline for comparison.

Ribeiro-Neto et al. developed three disk-based distributed algorithms to build global partitioned inverted files for large text collections [68]. The one that achieves the best results is the RR algorithm partially described in Section 5.2.1. After the computers receive the hits, they perform multiway merges to produce a run and afterwards build an inverted file with it. Using 8 computers interconnected with an 80 Mbps network, the RR algorithm built a distributed inverted file for the 100 GB TREC-7 text collection in 12 hours. With twice the computers, it built the

system	CPU	collection			index		time	speed	
	#	size GB	type	dcp.	part.	cp.	(hr)	1	2
Sidra	4	313.6	Web	yes	global	yes	56.38	1.39	16.2
RR	16	100	text	?	global	yes	6	1.04	?
Google	4	147.8	Web	yes	local	yes	147.4	0.25	13.5
CobWeb	312	500,000	Web	?	?	?	130.6	12.27	75

Table 5.1: Distributed systems to build large scale inverted files. **collection dcp.** indicates if the documents were decompressed before indexed. **index part.** indicates the type of partition used. **index cp.** indicates if the inverted index was compressed. **speed 1** is a measure of the number of GB indexed per CPU in one hour. **speed 2** is a measure of the number of pages indexed per CPU in one second.

index in half of the time, demonstrating a perfect speedup.

Only a few articles describe indexing of large Web scale collections, all using local partitioning schemes. This is simpler than creating a distributed index with a global partition, because after distributing the documents by all computers, it isn't necessary to exchange further information, except if global statistics for ranking computation are necessary (e.g. as the inverse document frequency). The initial Google system created a forward index (an index document-terms) of 147.8 GB of data from a crawl of 24 million Web pages, at a speed of 54 pages per second [15]. This totalizes 123.4 hours. The transformation of this forward index into an inverted index took an additional 24 hours using 4 computers. The same number of computers was assumed as the one used for the whole process.

CobWeb is the Internet Archive indexer and ranking system [62]. Its architecture is not published, only a few results are available from a slide show published in their Web site. It indexed more than 11 billion pages (0.5 PB of data) from the Internet Archive, the larger Web known. A cluster of 312 computers with 512 MB of memory and 500 GB of disk each, has an indexing speed of 75 pages per computer in each second. The Internet Archive Web is reported to be indexed by this cluster in 130.6 hours. The index has 2 Terabytes and grows sublinearly.

Table 5.1 shows that Sidra has performance results comparable to other systems, when the size of the indexed collection is on the order of 100-300 GB and the number of processors until 16. CobWeb operates on a completely different range: one order of magnitude more processors, a collection three orders of magnitude larger, and an indexing speed one order of magnitude faster. However, the algorithms and software architecture of this indexing system are not publicly documented.

Sidra provides speedup and scaleup characteristics which enables it to built indexes for very large Web collections as fast as necessary, by only adding more computers to the processing. This motivates the following discussion of the need to support index updates on Web search engines.

5.4 Index Updates

Since Web contents change continuously, there is a need to refresh the indexes of Web search engines frequently. Cho and Garcia-Molina analyzed the evolution of the Web during 4 months and discovered that 23% of the pages change every day, 15% change between a day and a week, and 16% change between a week and a month [27]. Fetterly et al. extended their work and presented results derived from eleven weeks of analysis [31]. During this time, 34.8% of the pages changed something, but the text of the pages only changed on around 10% of the pages. Given these conditions, it seems that an indexing system could profit from performing partial updates to indexes, instead of full reconstruction.

However, partial updates have their own problems:

1. the complexity of the indexing system increases, since indexes have to respond to queries, while being updated with new versions of pages at the same time.

2. index sizes increase due to internal fragmentation in the posting lists. Most techniques to update indexes reserve space apriori in the posting lists to add new postings in the future [16, 84]. In the majority of the systems, this also causes a degradation of query performance, because partial updates of indexes link existing posting lists with new ones on different disk blocks. As postings lists are not continuous in disk, i/o time and, consequently, the overall time increase.

3. for an efficient update, it is necessary a forward index of the collection. This has about the same size of the inverted file.

4. contrary to the expected, Cho and Garcia-Molina showed that for monthly crawls, partial updates provide indexes with freshness similar in average to the ones completely rebuilt [27].

Given this knowledge of previous Web studies and the performance observed with Sidra, it does not seem profitable to support partial index updates. Sidra enables fast creation of indexes for specific collections with pages changing at a higher frequency. Sidra can also support queries to multiple collections on the same computers, enabling it to support specialized indexes. This is also the path followed by some Web search engines. Google crawls and indexes from scratch around 4 billion pages about once a month (see `http://www.google.com/webmasters/2.html`), and offers a specialized search engine for news (see `http://news.google.com`), updated daily. A mechanism to search indexes created with different periodicities in parallel will be implemented in the future. Only the fresher versions of the documents indexed will be ranked and presented to the user.

5.5 Conclusion

Web collections are growing at a fast rate. Indexing systems need an efficient architecture and algorithms with scalability characteristics to keep up with this growth. This Chapter presented Sidra's architecture and algorithms developed for the creation of several indexes over large scale Web collections.

Sidra combines algorithms and techniques previously developed for classic IR systems, databases and Web search engines, to produce a high performance system capable of scaling the index creation of large scale collections of Web documents. These properties were validated with the indexing of the Portuguese Web, currently searchable on the tumba! Web search engine. A 313.6 GB Web collection was also indexed by Sidra in 56.38 hours using a cluster of 4 computers.

Chapter 6

Conclusions and Future Work

Web collections and Web users are growing at a fast rate. It is difficult for Web search engines to keep up offering the same response times and retrieval accuracy. This thesis presented the architectural design implementation and evaluation of Sidra, a new indexing, searching and ranking system for large scale Web collections. Sidra brought the tumba! Web search engine to performance levels comparable to those of state-of-the-art global Web search engines. Five initial objectives were proposed: high performance searching and indexing times, good quality of results, scalability and high service availability. Sidra includes mechanisms to accomplish all these objectives.

Sidra has a flexible architecture that enables the deployment of different configurations to respond to the needs of different Web search engines, while supporting load balancing and fault tolerance. It combines several algorithms and techniques applied on traditional IR systems, databases and Web search engines, to produce a scalable and high performance system to discover information on large Web scale collections of documents. It supports keyword searching, Boolean operators, phrase and "site:" searches in parallel.

The scalability and performance of the Sidra implementation was evaluated

over a realistic set of queries extracted from tumba!'s logs, and the Web collection indexed by tumba!. Tests were applied with different parameters to multiple Sidra configurations. Results showed that the system is scalable and provides high performance times. Using a cluster of 4 computers, Sidra has query response times of 100 milliseconds with a workload of 50 requests. The same system indexes the data and metadata from a 313.6 GB Web collection in 56.38 hours.

A ranking framework conceived to develop, test and evaluate ranking algorithms was built. It was used to evaluate the tumba!'s ranking quality supported by Sidra, following an evaluation methodology similar to the one used on the TREC Web Track. Results showed that the Sidra ranking framework enabled tumba! to reach further a quality close to Google on named entity recognition tasks. This framework is being used to study and improve the ranking quality of tumba!

Sidra is in operation on tumba! as the indexing, searching and ranking of tumba! since September 2003. Sidra has 20 thousand lines of code written essentially in C++ (the core of the system) and JAVA. Presently it responds to up to 20,000 queries/day on a collection of 3.2 million Web documents. The feedback from the users is very encouraging and some suggestions from them were incorporated in Sidra as new software releases were produced.

6.1 Future Work

Sidra was designed to support queries over multiple indexes of different types of information. A database is being built with information from automated document categorization, entity recognition and geographic location recognition software [76]. In the future, this information about the documents can be used to build specialized indexes for Sidra, enabling it to adapt results to query contexts.

Ranking algorithms are still being developed in a continuous process to im-

prove the quality of results. Some years ago, most people were happy with the results offered, based only in similarity ranking functions between the query and the document terms. Google and their PageRank algorithm raised the standards about the quality of results. I expect that this framework helps to raise even more these standards through the development of new algorithms, and the combination of existent ones that have shown successful results but are not implement in Sidra, such as Okapi BM25 [72].

The Web repository of tumba! enables the archive of successive versions of crawls. An objective for the future is to implement a mechanism that will enable the search of documents restricted to a time period, over the versions of documents stored over time. Another objective, is to implement a mechanism that will search indexes of different periodicities in parallel, for instance one with the pages that change every day (e.g. daily newspapers) and other not so dynamic. Only the fresher versions of the indexed documents would be ranked and presented to the user.

And to conclude, there is also the desire to test Sidra with larger Web collections and a larger number of computers. The results achieved were good, but a proper analysis of scalability would require a larger configuration and a larger data set. TREC provides for its new Terabyte Track, a collection of approximately 1 terabyte of Web pages that could be used for this analysis [7].

Bibliography

[1] Cross Language Evaluation Forum (CLEF). http://clef-qa.itc.it/2004.

[2] Direct Hit Web search engine. http://www.directhit.com.

[3] Jakarta Lucene - text search engine library. http://jakarta.apache.org/lucene.

[4] Oracle InterMedia. http://www.oracle.com/technology/products/intermedia.

[5] RFC 1305 - Network Time Protocol (version 3) specification, implementation. http://www.faqs.org/rfcs/rfc1305.html.

[6] Text REtrieval Conference (TREC). http://trec.nist.gov/.

[7] TREC Terabyte Track. http://www-nlpir.nist.gov/projects/terabyte.

[8] TREC Web Track. http://es.csiro.au/TRECWeb.

[9] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. ACM Press, 1999.

[10] Peter Bailey and David Hawking. A parallel architecture for query processing over a terabyte of text. Technical Report TR-CS-96-04, 1996.

[11] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: the Google cluster architecture. *IEEE Micro Magazine*, pages 22–28, March/April 2003.

[12] Michael K. Bergman. The deep Web: surfacing hidden value. *The Journal of Electronic Publishing from the University of Michigan*, 7, August 2001.

[13] Tim Berners-Lee, Robert Cailliau, Jean-Francois Groff, and Bernd Pollermann. World-Wide Web: the information universe. *Electronic Networking: Research, Applications and Policy*, 1(2):74–82, 1992.

[14] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[15] 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.

[16] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB)*, pages 192 – 202, September 1994.

[17] Eric W. Brown, James P. Callan, W. Bruce Croft, and J. Eliot B. Moss. Supporting full-text information retrieval with a persistent object store. In *Proceedings of the 4th Internationl Conference on Extending Database Technology–EDBT'94*, pages 365–378, 1994.

[18] Nicolas Bruno, Luis Gravano, and Amelie Marian. Evaluating top-k queries over Web-accessible databases. In *Proceedings of the 18th International Conference on Data Engineering*, April 2002.

[19] Chris Buckley and Alan F. Lewit. Optimization of inverted vector searches. In *Proceedings of the 8th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 97 – 110, 1985.

[20] Chris Buckley and Ellen M. Voorhees. Evaluating evaluation measure stability. In *Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 33–40, 2000.

[21] Forbes J. Burkowski. Retrieval performance of a distributed text database utilizing a parallel processor document server. In *Proceedings of the 2nd International Symposium on Databases in Parallel and Distributed Systems*, pages 71–79, 1990.

[22] Brendon Cahoon, Kathryn S. McKinley, and Zhihong Lu. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *ACM Transactions on Information Systems*, 18(1):1–43, 2000.

[23] James P. Callan, W. Bruce Croft, and Stephen M. Harding. The Inquery retrieval system. In *Proceedings of the 3rd International Conference on Database and Expert Systems Applications*, pages 78–83, 1992.

[24] João Campos and Mário J. Silva. Versus: a model for a Web repository. *CRC'01 - 4ª Conferência de Redes de Computadores*, November 2001.

[25] Nuno Cardoso, Mário J. Silva, and Miguel Costa. The XLDB group at CLEF 2004. In *CLEF 2004 - Multilingual Question Answering Evaluation*, 2004.

[26] David Carmel, Doron Cohen, Ronald Fagin, Eitan Farchi, Michael Herscovici, Yoëlle S. Maarek, and Aya Soffer. Static index pruning for information retrieval systems. In *Proceedings of the 24th ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 43–50, 2001.

[27] Junghoo Cho and Hector Garcia-Molina. The evolution of the Web and implications for an incremental crawler. In *Proceedings of the 26th Inter-*

national Conference on Very Large Databases, pages 200–209, September 2000.

[28] Miguel Costa and Mário J. Silva. Ranking no motor de busca TUMBA. *CRC'01 - 4ª Conferência de Redes de Computadores*, November 2001.

[29] David DeWitt and Jim Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[30] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 102–113, May 2001.

[31] Dennis Fetterly, Mark Manasse, Marc Najork, and Janet L. Wiener. A large-scale study of the evolution of Web pages. In *Proceedings of the 12th International World Wide Web Conference, WWW2003*, pages 669–678, May 2003.

[32] Ophir Frieder, Abdur Chowdhury, David Grossman, and M. Catherine McCabe. On the integration of structured data and text: A review of the SIRE architecture. In *DELOS Workshop on Information Seeking, Searching and Querying in Digital Libraries*, December 2000.

[33] Ophir Frieder and Hava Tova Siegelmann. On the allocation of documents in multiprocessor information retrieval systems. In *Proceedings of the 14th Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval*, pages 230–239, 1991.

[34] Daniel Gomes. Tarântula - sistema de recolha de documentos na WWW. Technical report, Faculdade de Ciências da Universidade de Lisboa, July 2001. Relatório do Estágio Profissionalizante da FCUL.

[35] Daniel Gomes, João Campos, and Mário J. Silva. Versus: a Web repository. *Workshop on Distributed Data & Structures (WDAS 2002)*, 2002.

[36] Daniel Gomes and Mário J. Silva. Tarântula - sistema de recolha de documentos da Web. *CRC'01 - 4ª Conferência de Redes de Computadores*, November 2001.

[37] Daniel Gomes and Mário J. Silva. A characterization of the Portuguese Web. In *Proceedings of 3rd ECDL Workshop on Web Archives*, August 2003.

[38] Torsten Grabs, Klemens Böhm, and Hans-Jörg Schek. PowerDB-IR: information retrieval on top of a cluster of databases. In *Proceedings of the 10th International Conference on Information and Knowledge Management*, November 2001.

[39] Ulrich Guntzer, Wolf-Tilo Balke, and Werner Kiessling. Optimizing multi-feature queries for image databases. In *Proceedings of the 26th International Conference on Very Large Databases*, pages 419–428, September 2000.

[40] Taher H. Haveliwala. Efficient computation of PageRank. Technical report, Stanford University, 1999.

[41] Taher H. Haveliwala. Efficient encodings for document ranking vectors. Technical report, Stanford University, December 2002.

[42] Taher H. Haveliwala. Topic-sensitive PageRank. In *Proceedings of the 11th International World Wide Web Conference*, 2002.

[43] David Hawking. Scalable text retrieval for large digital libraries. In *European Conference on Digital Libraries*, pages 127–145, 1997.

[44] Monika R. Henzinger, Rajeev Motwani, and Craig Silverstein. Challenges in Web search engines. *SIGIR Forum*, 36(2):11–22, September 2002.

[45] Bernard J. Jansen, Amanda Spink, Judy Bateman, and Tefko Saracevic. Searchers, the subjects they search, and sufficiency: a study of a large sample of Excite searches. In *Proceedings of the World Conference on the WWW and Internet*, 1998.

[46] 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.

[47] Byeong-Soo Jeong and Edward Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):142–153, 1995.

[48] Byeong-Soo Jeong and Edward Omiecinski. Index file partitioning in parallel database systems. *Submitted to the International Conference on Data Engineering*, 1996.

[49] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[50] W. Kraaij, T. Westerveld, and D. Hiemstra. The importance of prior probabilities for entry page search. In *Proceedings of the 25th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 27–34, 2002.

[51] Xiaohui Long and Torsten Suel. Optimized query execution in large search engines with global page ordering. In *Proceedings of the 29th International Conference on Very Large Databases*, pages 129–140, September 2003.

[52] Bruno Martins and Mário J. Silva. WebCAT: A Web content analysis tool for IR applications. To be submitted, 2004.

[53] 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.

[54] Alistair Moffat. Resource-limited index construction for large texts. In *Proceedings of the 17th Australasian Computer Science Conference*, pages 169–178, 1994.

[55] Alistair Moffat and Lang Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, July 2000.

[56] Alistair Moffat and Justin Zobel. Fast ranking in limited space. In *Proceedings of the 10th International Conference on Data Engineering*, pages 428–437, February 1994.

[57] James K. Mullin. Optimal semijoins for distributed database systems. *IEEE Transactionson Software Engineering*, 16(5):558–560, May 1990.

[58] Surya Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *Proceedings of the 15th International Conference on Data Engineering*, pages 22–29, March 1999.

[59] Michael Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of USENIX Technical Conference, FREENIX Track*, June 1999.

[60] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: bringing order to the Web. Technical report, Stanford Digital Library Technologies Project, 1998.

[61] Gopal Pandurangan, Prabhakara Raghavan, and Eli Upfal. Using PageRank to characterize Web structure. In *Proceedings of the 8th International Conference on Computing and Combinatorics*, pages 330 – 339, 2002.

[62] Anna Patterson. Cobweb search. http://ia00406.archive.org/cobwebsearch.ppt, 2004.

[63] Michael Persin. Document filtering for fast ranking. In *ACM SIGIR Conference*, pages 339–349, August 1994.

[64] Roger S. Pressman. *Software Engineering: a Practitioner's Approach*. McGraw-Hill Computer Science Series, 1997.

[65] Raghu Ramakrishnan. *Database Management Systems*. McGraw-Hill Computer Science Series, 1998.

[66] Patrick Reynolds and Amin Vahdat. Efficient peer-to-peer keyword searching. Unpublished manuscrit, February 2002.

[67] Berthier Ribeiro-Neto and Ramurti A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the 3rd ACM Conference on Digital libraries*, pages 182–190, June 1998.

[68] Berthier Ribeiro-Neto, João Paulo Kitajima, Gonzalo Navarro, Cláudio Sant'Ana, and Nivio Ziviani. Parallel generation of inverted files for distributed text collections. In *Proceedings of the 18th International Conference of the Chilean Computer Science Society*, November 1998.

[69] Berthier Ribeiro-Neto, Edleno S. Moura, Marden S. Neubert, and Nivio Ziviani. Efficient distributed algorithms to build inverted files. In *Proceedings of the 22th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 105–112, August 1999.

[70] Mathew Richardson and Pedro Domingos. The intelligent surfer: probabilistic combination of link and content information in PageRank. In *Advances in Neural Information Processing Systems 14*. MIT Press, 2002.

[71] Knut Magne Risvik and Rolf Michelsen. Search engines and Web dynamics. *Computer Networks*, 39:289–302, June 2002.

[72] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In *Proceedings of the Text REtrieval Conference*, pages 109–127, 1995.

[73] Gerard Salton. *Introduction to Modern Information Retrieval*. McGraw-Hill Computer Science Series, 1983.

[74] Robert Sedgewick. *Algorithms in C*. Addison Wesley, 1990.

[75] Mário J. Silva. The case for a Portuguese Web search engine. In *Proceedings of the IADIS WWW/Internet 2003 Conference*, November 2003.

[76] Mário J. Silva, Bruno Martins, Marcirio Chaves, Nuno Cardoso, and Ana Paula Afonso. Adding geographic scopes to Web resources. In *Workshop on Geographic Information Retrieval, SIGIR 2004*, 2004.

[77] Mário J. Silva, Bruno Martins, and Miguel Costa. *Avaliação Conjunta de Recuperação de Informação da Web Portuguesa*, chapter 11. 2005.

[78] Craig Silverstein, Monika Henzinger, Hannes Marais, and Michael Moricz. Analysis of a very large Altavista query log. Technical Report 1998-014, Digital Systems Research Center, 1998.

[79] Amanda Spink, Seda Ozmutlu, Huseyin C. Ozmutlu, and Bernard J. Jansen. U.S. versus European Web searching trends. *SIGIR Forum*, 36(2):32–38, 2002.

[80] Craig Stanfill and Brewster Kahle. Parallel free-text search on the connection machine system. *Communications of the ACM*, 29(12):1229–1239, December 1986.

[81] Craig Stanfill, Robert Thau, and David Waltz. A parallel indexed algorithm for information retrieval. In *Proceedings of the 12th Annual International Conference on Research and Development in Information Retrieval*, pages 88–97, June 1989.

[82] Anthony Tomasic and Hector Garcia-Molina. Caching and database scaling in distributed shared-nothing information retrieval systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 129–138, May 1993.

[83] Anthony Tomasic and Hector Garcia-Molina. Performance of inverted indices in distributed text document retrieval systems. In *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems*, pages 8–17, 1993.

[84] Anthony Tomasic, Hector Garcia-Molina, and Kurt A. Shoens. Incremental updates of inverted lists for text document retrieval. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 289–300, May 1994.

[85] Trystan Upstill, Nick Craswell, and David Hawking. Predicting fame and fortune: PageRank or Indegree? In *Proceeding of the 8th Australasian Document Computing Symposium*, December 2003.

[86] Trystan Upstill, Nick Craswell, and David Hawking. Query-independent evidence in home page finding. In *Proceedings of the ACM Transactions on Information Systems (TOIS)*, volume 21, July 2003.

[87] Ellen M. Voorhees. The TREC-8 question answering track report. In *Proceedings of TREC-8*, November 1999.

[88] Ellen M. Voorhees and Chris Buckley. The effect of topic set size on retrieval experiment error. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 316–323, 2002.

[89] Thijs Westerveld, Wessel Kraaij, and Djoerd Hiemstra. Retrieving Web pages using content, links, urls and anchors. In *TREC-2001 Notebook Proceedings*, 2001.

[90] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and indexing documents and images*. Morgan Kaufmann, 1994.

[91] Wai Yee Peter Wong and Dik Lun Lee. Implementations of partial document ranking using inverted files. *Information Processing and Management*, 29(5):647–669, May 1992.

[92] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, 1998.