A Hybrid Scavenger Grid Approach to Intranet Search

by

Ndapandula Nakashole

Supervised by: Dr Hussein Suleman

Thesis presented for the Degree of Master of Science

In the Department of Computer Science

University of Cape Town

· February 2009 ·

I know the meaning of plagiarism and declare that all of the work in the document, save for which is properly acknowledged, is my own.

Ndapandula Nakashole

"An avalanche of content will make context the scarce resource. Consumers will pay serious money for anything that helps them sift and sort and gather the pearls that satisfy their fickle media hungers. The future belongs to neither the conduit or content players, but those who control the filtering, searching and sense-making tools."

—PAUL SAFFO (1994)

"We have become far more proficient in generating information than we are in managing it, and we have also built technology that easily allows us to create new information without human intervention."

—JONATHAN B. SPIRA AND DAVID M. GOLDES (2007)

Acknowledgements

I would like to extend my heartfelt thanks to my supervisor for his guidance and support through every step of this thesis. The discussions I had with him have not only shaped this thesis into the document it is but they have also inspired my growth as a graduate student.

I thank the members of the Digital Library and High Performance Computing laboratories for interesting paper discussions. I am grateful for the support and kindness of my family and Ming Ki Chong.

The financial assistance of the National Research Foundation(NRF) towards this research is hereby acknowledged. Opinions expressed, and conclusions arrived at, are those of the author and are not to be attributed to the NRF.

Abstract

According to a 2007 global survey of 178 organisational intranets, 3 out of 5 organisations are not satisfied with their intranet search services. However, as intranet data collections become large, effective full-text intranet search services are needed more than ever before. To provide an effective full-text search service based on current information retrieval algorithms, organisations have to deal with the need for greater computational power. Hardware architectures that can scale to large data collections and can be obtained and maintained at a reasonable cost are needed. Web search engines address scalability and cost-effectiveness by using large-scale centralised cluster architectures. The scalability of cluster architectures is evident in the ability of Web search engines to respond to millions of queries within a few seconds while searching very large data collections. Though more cost-effective than high-end supercomputers, cluster architectures still have relatively high acquisition and maintenance costs. Where information retrieval is not the core business of an organisation, a cluster-based approach may not be economically viable.

A hybrid scavenger grid is proposed as an alternative architecture — it consists of a combination of dedicated and dynamic resources in the form of idle desktop workstations. From the dedicated resources, the architecture gets predictability and reliability whereas from the dynamic resources it gets scalability. An experimental search engine was deployed on a hybrid scavenger grid and evaluated. Test results showed that the resources of the grid can be organised to deliver the best performance by using the optimal number of machines and scheduling the optimal combination of tasks that the machines perform. A system-efficiency and cost-effectiveness comparison of a grid and a multi-core machine showed that for workloads of modest to large sizes, the grid architecture delivers better throughput per unit cost than the multi-core, at a system efficiency that is comparable to that of the multi-core.

The study has shown that a hybrid scavenger grid is a feasible search engine architecture that is cost-effective and scales to medium- to large-scale data collections.

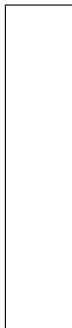


Table of Contents

1 Introduction **1**

1.1 What is a Hybrid Scavenger Grid? 3

1.1.1 Resource Availability 4

1.1.2 Applicability to Resource Constrained Environments 4

1.2 Aims 5

1.3 Methodology 6

1.4 Thesis organisation 7

2 Background and Related Work **9**

2.1 Information Retrieval 10

2.1.1 The Retrieval Process 10

2.1.2 Retrieval Models 11

2.1.3 Index Construction 14

2.1.4 Index Maintenance 19

2.1.5 Web Information Retrieval 20

2.2 Grid Computing 21

2.2.1 Grid Architecture 22

2.2.2 Global Computing Grids 23

2.2.3 Institutional Grids 24

2.2.4 Global Computing vs Institutional Grids 24

2.2.5 The Globus Toolkit 25

2.3 Related Work 26

2.3.1 Intranet Search 26

2.3.2 Grid-based Information Retrieval 27

2.3.3 Issues in Distributed Information Retrieval 29

2.4 Summary 31

3 Condor and SRB Grid Middleware **33**

3.1 Condor 33

3.1.1 Overview . . . 33
3.1.2 Condor Setup . . . 34
3.2 The Storage Resource Broker . . . 36
3.2.1 Overview . . . 36
3.2.2 SRB Setup . . . 38
3.3 Summary . . . 39

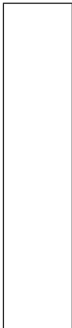
4 Design and Implementation 41
4.1 Search Engine Architecture . . . 41
4.2 Inter-Component Communication . . . 42
4.2.1 User Interface to Scheduler Communication . . . 42
4.2.2 Scheduler to Worker Agents Communication . . . 43
4.3 The User Interface . . . 44
4.4 The Scheduler . . . 44
4.4.1 Node Starter . . . 45
4.4.2 Data Manager . . . 45
4.4.3 The Query Handler . . . 47
4.4.4 Result Merger . . . 47
4.5 Roles of the Nodes . . . 48
4.6 Search Engine Jobs . . . 49
4.6.1 Indexing Job . . . 49
4.6.2 Querying Job . . . 53
4.7 Summary . . . 54

5 Evaluation 55
5.1 Resources . . . 56
5.1.1 Data Sets . . . 56
5.1.2 Query Set . . . 57
5.1.3 Hardware and Performance Measurement utilities . . . 59
5.2 Grid Middleware Overhead . . . 61
5.2.1 SRB Throughput . . . 61
5.2.2 SRB Impact on Indexing . . . 63
5.3 Varying Grid Composition . . . 65
5.3.1 Varying Dynamic Indexers . . . 65
5.3.2 Index Merging . . . 71
5.3.3 Utilizing Storage Servers . . . 72
5.4 Index Maintenance . . . 75
5.5 Hybrid Scavenger Grid versus Multi-Core . . . 78
5.6 Querying Performance Analysis . . . 85
5.7 Summary . . . 87

6 Conclusion 89

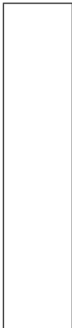
References 93

A Appendix: Queries used in the Experiments 99



List of Tables

5.1 Summary of the text collections used in the experiments. The AC.UK collection is the result of a crawl of the domain of academic institutions in the UK. The UCT.AC.ZA collections are partial snapshots of the `uct.ac.za` domain taken a month apart. 57
5.2 Query set summary. Queries are extracted from the Google Insights for Search service. 59
5.3 Sample queries extracted from Google Insights for Search in the United Kingdom region for the time frame "2004–November 2008". 60
5.4 SRB read and write speeds. 62



List of Figures

2.1 The retrieval process carried out by an information retrieval system 11
2.2 Example of the inverted index construction process. 16
2.3 The resulting inverted index. For clarity posting lists only show document identifiers. 17
2.4 Typical architecture of a Web search engine. 21
2.5 A conceptual view of the grid. 22
2.6 The Grid layered architecture. 23
2.7 Virtual Organisations enable organisations or departments to share resources. . . . 24
2.8 The GridIR proposed architecture. 28

3.1 Daemon layout as deployed in the grid used in this thesis. In this case the grid is busy and all the nodes are running jobs. 36
3.2 SRB storage abstraction. 37
3.3 SRB setup in the grid. 38
3.4 Steps involved in retrieving data from SRB. 39

4.1 High level components of the experimental search engine architecture. These are: User Interface, Scheduler, Condor, Worker Agents and SRB. 42
4.2 The Query Handler part of the Scheduler is exposed as a Web service. The Scheduler passes queries on to the Worker Agents via socket connections. 43
4.3 Components of the Scheduler. 44
4.4 Example Condor job description file to start Worker Agents on 3 dynamic nodes for the task of indexing 46
4.5 Example Condor description file to start Worker Agents on 2 storage servers (dedicated nodes) for the task of querying 47
4.6 Components of the dedicated and dynamic node Worker Agents. 49
4.7 The Non-Local Data and Local Data distributed indexing approaches. 51
4.8 Query processing at a storage server. 53

5.1 Top searches on Google Insights for Search. The results returned are for the Web search volume of the period 2004–November 2008 for the United kingdom region in the Physics category. 58

5.2 Computer systems used for the experiments. The computer systems are within a 100 Mbps network. The Linux cluster computers are interconnected by a Gigabit Ethernet network. 60
5.3 Network connection of the nodes. machine7 and machine2 are within a Gigabit Ethernet network with a measured maximum bandwidth of 75 MB/sec. nut.cs is connected to machine2 by a 100 Mbps network with a measured maximum bandwidth of 12 MB/sec. 62
5.4 Local Data and Non-local Data indexing 64
5.5 Indexing performance benchmarks. local Lucene refers to plain non-distributed Lucene, dedicated refers to one of the dedicated nodes and dynamic refers to one of the dynamic nodes. 64
5.6 Indexing performance for dynamic indexers. The approach evaluated is Local Data indexing, using 6 SRB storage servers. 67
5.7 System efficiency during indexing 70
5.8 Indexing time inclusive and non-inclusive of merging time 72
5.9 Performance of storage servers. The data size used is 128 GB. 74
5.10 Job completion comparison for the case Indexing + Indices and the case of Source files + Indexing + Indices. The Data size used is 128 GB, 6 storage servers were used in both cases. 75
5.11 Performance of storage servers. Data size used is 128 GB, 6 and 9 storage servers used. 76
5.12 Index maintenance performance. The mergefactor controls the number of documents buffered in memory. 77
5.13 Performance, system efficiency and cost-effectiveness: 32 GB 81
5.14 Performance, system efficiency and cost-effectiveness: 128 GB 82
5.15 Performance, system efficiency and cost-effectiveness: 256 GB 83
5.16 Querying performance with 6 storage/query servers. 86

CHAPTER

1 Introduction

INFORMATION overload has been a concern for a long time. In 1975, it was estimated that about 50 million books had been published [41]. A study conducted in 2000 [57] estimated that the world produces between 1 and 2 exabytes[1] of unique information per year. These statistics might be estimates but they serve to underscore an existing problem — information is increasing in quantity faster than it can be managed.

Today the problem persists, with data being produced at faster rates to fill space made available by the constantly expanding and ever cheaper storage. More and more users are becoming active on the Web, not only consuming but also producing information. As of December 2007, the blog search engine Technorati [86] was tracking more than 112 million blogs. These blogs belong to individuals or groups who regularly publish content on the Web through blog posts. In addition, medical records of entire nations are being digitised. The British national health service is in the process of implementing one such effort [67]. The United States has called for large scale adoption of electronic medical records with the goal of covering all of its citizens by 2014 [17]. Furthermore, scholarly publications are made available electronically as academic and research institutions adopt open access institutional repositories, using tools such as EPrints [24], in order to maximise their research impact. All these contribute to information explosion. A large part of this information is part of the Web. The challenge is to preserve while providing easy and fast access to this information.

In recent years, Web search engines like Yahoo! and Google have enabled users on the Web to efficiently search for documents of interest. Results are returned in a few seconds, with potentially relevant documents ranked ahead of irrelevant ones. These technology companies compete with one another to provide high quality search services requiring complex algorithms and vast

[1] An exabyte is a billion gigabytes or 10^{18} bytes.

computer resources, at no direct financial cost to the user.

The situation is different within organisational intranets where search services are inefficient if they are present at all [65] . Many times the sentiment is "why not just use Google?". Firstly, not all data within intranets is publicly available and thus cannot be reached by Web search engine spiders that download data for indexing. Some data — for example, design documents, confidential company documents and patient records — is private and cannot be entrusted to external parties for indexing. Secondly, users within intranets look for specific information, for example a definition of a term, or a person who has certain duties. In such cases results from a generic Web search engine may not be adequate.

Much effort has been made in both industry and academia to address intranet search requirements. Commercial software systems dedicated to intranet search have been developed — examples include: FAST Enterprise Search [27], Autonomy [6], OmniFind [68] and Microsoft Sharepoint [62]. The focus of these software tools is not on the hardware required to power the search system. It is up to the organisation to determine the hardware infrastructure with the storage and computational power to deliver the desired performance. The Google Search Appliance [40] is a hardware device that provides intranet search and is able to handle up to millions of pages. However, this device has initial acquisition costs and it is a black-box solution with no option to tune it for the specific needs of the organisation.

Many large-scale Web service providers — such as Amazon, AOL, Google, Hotmail, and Yahoo! — use large data centres, consisting of thousands of commodity computers to deal with their computational needs. The Google search engine architecture combines more than 15,000 commodity-class PCs with fault-tolerant software [9]. The key advantage of this architectural approach is its ability to scale to large data collections and millions of user requests. This is evident in Web search engines which respond to millions of queries per day. Clusters of commodity computers are known for their better cost/performance ratio in comparison to high-end supercomputers. However, there is still a high cost involved in operating large data centres. Such data centres require investment in a large number of dedicated commodity PCs. In addition, they need adequate floor space, cooling and electrical supplies. IDC[2] reported that in 2007 businesses spent approximately $1.3 billion to cool and power spinning disk drives in corporate data centres and this spending is forecasted to reach $2 billion in 2009[19].

If information retrieval is not the core business of the organisation, it may be difficult to justify expenditure on a data centre. In addition, if an organisation does not need the computers for other tasks, they may not be highly utilised at all times. Furthermore, it is not clear how much more data centres can be scaled up at a reasonable cost if both data and workload continue to

[2] International Data Corporation (IDC) is a market research and analysis firm specialising in information technology, telecommunications and consumer technology. http://www.idc.com.

grow in the coming years. Baeza-Yates et al.[8] estimate that, given the current amount of Web data and the rate at which the Web is growing, Web search engines will need 1 million computers in 2010. It is therefore important to consider other approaches that can cope with current and future growth in data collections and be able to do so in a cost-effective manner.

This thesis proposes an alternative architecture — a hybrid scavenger grid consisting of both dedicated servers and dynamic resources in the form of idle workstations to handle medium- to large-scale [3] search engine workloads. The dedicated resources are expected to have reliable and predictable behaviour. The dynamic resources are used opportunistically without any guarantees of availability. These dynamic resources are a result of unused capacity of computers, networks and storage within intranets. Employing these two types of resources to form a hybrid architecture has a number of advantages not available with centralised data centres. A hybrid architecture provides cost-effective scalability and high utilisation of the dedicated servers.

The main objective of this research is to investigate a way to deliver scalable search services by means of a hybrid scavenger grid consisting of dedicated and dynamic resources. The investigation covers choice of job scheduling and distributed storage management middleware, system performance analysis, and a cost/performance analysis of centralised versus distributed architectures. Experimental results highlight the benefits of augmenting dedicated resources with opportunistic idle desktop workstations in delivering scalable search services.

1.1 WHAT IS A HYBRID SCAVENGER GRID?

A "scavenger" or "cycle-scavenger" grid is a distributed computing environment made up of under-utilised computing resources in the form of desktop workstations, and in some cases even servers, that are present in most organisations. Cycle-scavenging provides a framework for exploiting these under-utilised resources, and in so doing providing the possibility of substantially increasing the efficiency of resource usage within an organisation. Global computing projects such as FightAIDS@Home [28] and SETI@Home [81] have already shown the potential of cycle-scavenging. The idea of a hybrid scavenger grid is an extension of cycle-scavenging— it adds the notion of dedicated and dynamic resources.

Dedicated computing resources are guaranteed to be available for search engine processing. These can be in the form of clusters, multi-processors or single CPU machines. Dynamic resources in the form of idle resources are used by the system on-demand as they become available and as required by the system's computational needs. Availability of dynamic resources depends on work patterns of the people within an organisation, thus dynamic resources are not reliable. Therefore, dedicated

[3] Medium- to large-scale refers to data ranging from hundreds of gigabytes to petabytes of data.

nodes are needed to provide search services that are reliable and have a predictable uptime. If dynamic nodes are exclusively used, for example, if a query is routed to a desktop machine (a dynamic resource) and the machine is reclaimed by the owner half-way through searching, it may mean delayed responses. During busy times of the day when dynamic nodes are not available for search engine operations, dedicated nodes can be used exclusively. Due to the limited number of dedicated nodes, they cannot provide the scalability required to deal with large data collections. Thus the dedicated nodes should be augmented with the large number of dynamic nodes that become available during non-working hours. From the dedicated nodes, the architecture gets reliability; from the dynamic nodes it gets scalability.

1.1.1 Resource Availability

The hybrid scavenger grid assumes that dynamic resources are readily available within organisations. The question to ask is whether organisations have enough desktop machines idle for long enough to handle information retrieval operations. The size of the organisation determines how much data it produces — typically the larger the organisation, the more data it produces. An organisation's growth usually includes an increase in the number of people and hence the number of desktop workstations. Typically, within organisations, desktop machines are busy during working hours. At night and on weekends, machines are generally left doing little to no work. Thus, work patterns within an organisation can be established to determine suitable times for scheduling jobs related to information retrieval such as text indexing.

1.1.2 Applicability to Resource Constrained Environments

In developing countries there are unique resource limitations such as limited computer resources and no high-speed interconnects. Organisations in the developing world have low IT budgets. Consequently, many organisations in Africa, including universities, have limited computer hardware resources. A survey on African higher education conducted by Harvard University in 2007 revealed that most students hoped that international donors could provide computers to their universities [44]. Students often have to share computers in under-equipped laboratories or libraries. A 2006 study on computer access in 13 federal universities in Nigeria found that on average a university had about 13 computers in its library. In one case a university with 30,000 undergraduates had 19 computers in its library [25].

Due to resource limitations in organisations in the developing world, maximum resource utilisation is a key concern. The hybrid scavenger grid ultimately supports utilising resources already at an organisation's disposal. Thus it has the potential to empower organisations in the developing world to provide their own search services with limited but well-utilised computing resources.

1.2 AIMS

The main research question of this thesis is whether it is beneficial to use a combination of dedicated and dynamic resources for the search engine operations of indexing and searching. Once a benefit is established, the next question is how these two types of resources can be organised in a way that delivers the best improvement in performance. Because of the distributed nature of the computing resources being used, the system should deal with distributed information retrieval issues including load balancing, query routing and result merging. In dealing with these issues the system is likely to incur additional overhead that can impact system performance. It is therefore important to ensure that such overheads do not affect overall search performance. The search engine should be able to respond to queries within a few seconds, preferably less, similar to what users of Web search engines are accustomed to.

More specifically, this research aims to answer the following questions:

1. Is it beneficial to use a hybrid scavenger grid architecture for indexing and searching?
   (a) Can indexing time be reduced?
   (b) Are sub-second query responses possible with this architecture?
2. How can such an architecture be organised in a way that delivers the best performance?
3. Is such an architecture cost-effective in comparison to alternatives?
4. Can such an architecture scale to large collections of data?

1.3 METHODOLOGY

This research involved developing an experimental search engine. The focus of this work is not on developing new information retrieval algorithms but rather on a different hardware architecture. Therefore, the developed prototype uses the Lucene [56] open source search engine as the underlying information retrieval engine. In addition, it uses the Condor job scheduler [20, 55] for job submission and tracking. For distributed data management, the system employs the Storage Resource Broker (SRB)[10, 73] data grid middleware which provides a layer of abstraction over data stored in various distributed storage resources, allowing uniform access to distributed data. The most recent generation of data grid middleware known as i Rule Oriented Data Systems (iRODS) was released when this project was already underway, hence iRODS was not used.

The glue that enables the three tools — Lucene, Condor and SRB — to work together, as well as additional system components, was implemented in the Java programming language in a Linux environment. The developed system has the ability to construct an index of a data collection by partitioning the data into manageable work chunks that can be reallocated in case of machine failure. The search engine also has the ability to update an existing index — taking into account additions, deletions and modifications of documents. In a distributed search engine, there are multiple query servers, each holding a partial index. For each query, each server retrieves a partial list of ranked documents based on their scores. These partial results are merged to create a single result set which is returned to the user. However, the scores from the query servers are not comparable because each server computes similarity scores between documents and queries using local collection statistics which may vary across collections. The search engine addresses the issue of incomparable scores by re-ranking the results. The search functionality was exposed as a Web service.

Tests were carried out to evaluate the system. The tests performed were grouped into five main categories. The first set of tests involved evaluating the overheard introduced by the presence of SRB grid middleware. The aim was to determine SRB data ingest rates as well as download rates and subsequently establish the impact of these rates on performance. The second set of tests involved testing indexing performance with the aim of determining how the dedicated and dynamic resources can be best organised to deliver improved indexing performance. The third set of tests evaluated index maintenance performance to determine the speed of updating the index. The fourth set of tests evaluated query performance with the aim of determining query throughput. The final set of tests was a comparison between two types of hardware architectures (Grid architecture vs. Multi-core) to determine which architecture achieves better job throughput per unit cost for a given workload.

1.4 THESIS ORGANISATION

Chapter 2 presents an overview of concepts of information retrieval and grid computing. A discussion on index construction and updating strategies is presented and an overview of related work in the area of intranet search and grid-based information retrieval is provided. **Chapter 3** is an in-depth coverage of the grid middleware used in the prototype developed. The chapter explains the configuration and setup of the grid middleware as used in the experimental system. **Chapter 4** discusses the design and implementation of the hybrid scavenger grid-based search engine. **Chapter 5** details the experiments carried out to evaluate the developed search engine. **Chapter 6** provides concluding remarks; it also puts forward some limitations of the architecture and possible future work directions.

Chapter 2

Background and Related Work

The ability to search over a text collection is highly desirable in the face of information overload. The main goal of information retrieval is to provide search over text collections. Consequently, information retrieval systems are an integral part of the majority of today's computer systems, with Web search engines being the most visible application of information retrieval. In addition to Web-based information retrieval, a wide range of non-Web-based systems with varying capabilities exist. These include enterprise search engines, for example FAST Enterprise Search [27] and OmniFind [68]; filesystem search tools, for example `locate` and `find` tools in UNIX and desktop search in both Microsoft Windows and Apple's Mac OS X; and search capabilities embedded within email clients.

In developing a full-text search system, special data structures are required in order to ensure efficiency. An overview of the process of constructing an index in the form of an inverted index — the de facto standard data structure for keyword-based search operations, is given in section 2.1.

Limitations in processor speeds and the costly nature of supercomputers, prompted the use of distributed computing for large scale information retrieval. Cluster and grid distributed computing paradigms have been studied and used as underlying architectures for information retrieval systems. A compute cluster is a type of distributed processing system which consists of stand-alone computers interconnected via a high speed network and working together as a single computing resource. Clusters usually contain a single type of processor and operating system, whereas grids can contain machines from different vendors running various operating systems. This thesis focuses on the grid computing paradigm. An overview of grid computing is given in section 2.2. A discussion of work already done in grid-based information retrieval is in section 2.3.

The chapter first presents an overview of relevant concepts in information retrieval and grid computing before discussing related work.

2.1 Information Retrieval

2.1.1 The Retrieval Process

An information retrieval system has two central components [2]: A *document collection* and an *information need*. A document collection (also referred to as a *corpus*) is a group of documents over which retrieval is performed. A document in this context refers to the basic unit over which the retrieval system is built; for example, it can be a Web page or a book chapter or a plain text file. An information need (also referred to as a *query*) is a topic which the user wants to know more about. Given these two components, the basis of an information retrieval system is to find documents with terms that match the terms in the query. Following this approach, an information retrieval system needs to be able to represent both documents and queries in a way that allows their fast comparison.

The retrieval system, as shown in Figure 2.1 adapted from [2], parses and transforms documents in the document collection into document representations which are structured descriptions of the information contained in the documents — a process known as *indexing* [58]. The end product of the indexing process is an *index*. The information retrieval system handles a user's query by comparing the terms in the query to those in the index. It then retrieves potentially relevant documents and ranks them such that documents near the top are the most likely to be what the user is looking for. A document is relevant if it addresses the information need, not necessarily because it contains all the words in the query. For example, a user types the term *Java* into a Web search engine, wanting to know more about Java the Indonesian island. The information retrieval system is likely to also return documents about the Java programming language, which do contain the word but do not address the information need. To handle this problem, some information retrieval systems include what is known as *relevance feedback* [37]. The idea of relevance feedback is to involve the user in the retrieval process in order to improve the final result set. The user gives feedback on the relevance of documents in an initial set of results. The system computes a better representation of the query based on the user feedback. Relevance feedback can be a burden to the user, thus, in practice, other forms of interactive retrieval, which require little to no effort from the user and are just as efficient, are preferred. For example Web search engines use *query expansion* [98] by suggesting related queries and the users choose one of query suggestions that is closer to their own information need.

The discussion in this section is a high level overview of the retrieval process without the specifics of how the index is constructed and how documents are matched with queries. The next few sections provide these details. Before discussing the inner details of an index and what data structures have been proposed for this purpose, first a description of what type of retrieval model

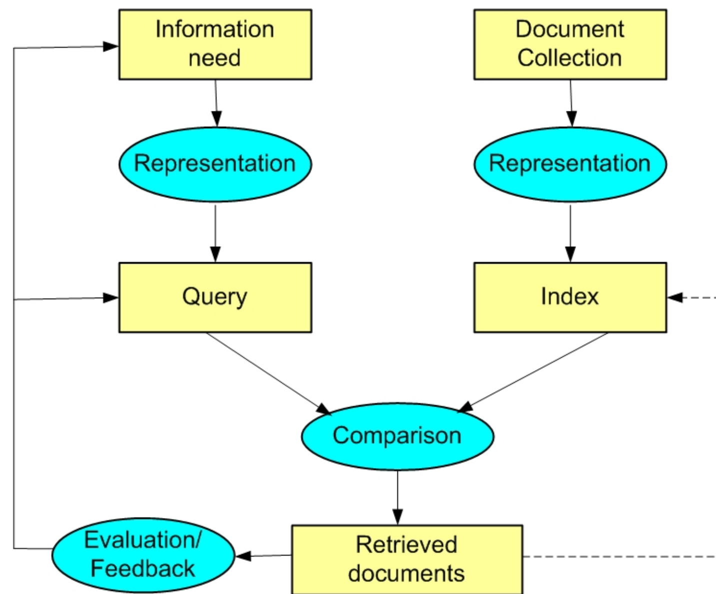


FIG. 2.1: The retrieval process carried out by an information retrieval system

the index needs to support is given.

2.1.2 Retrieval Models

A retrieval model [2, 58] is an abstraction of the retrieval process whose purpose is to describe the computation process of a retrieval system. Retrieval models are classified into those that provide exact-match and those that provide a best match of queries to documents.

In exact-match retrieval models, a query is precise and every document either matches or it does not. The result set from an exact-retrieval model is a listing of documents that exactly match the query with no ordering of documents [2]. For example, relational databases provide exact match results to queries. On the other hand, a query in best-match retrieval models describes "best" matching documents, thus every document matches the query to some degree. In the best-match model, the result set is a ranked list of documents. For example, Web search engines provide best-match results, often returning several pages of results per query listed in decreasing order of estimated relevance.

While exact-match approaches have the advantage that they can be implemented efficiently, they have disadvantages too: Firstly, formulation of queries is not easy unless users know the collection well and know what they are looking for, which in practice is often not the case. Secondly, result sets are often unacceptably short, only including documents that are exact matches. Best-match approaches are significantly more effective than exact-match alternatives. They ,however, have the downside that it is more difficult to formulate an appropriate model and that they often provide

less efficiency than exact-match models. Details about models based on each type of retrieval are given below.

Boolean Model

The Boolean model [36] is an exact-match model. In a Boolean model, a document is represented as a set of terms that appear in that document. Boolean queries are in the form of a Boolean expression of terms. That is, terms are combined using Boolean operators: AND, OR and NOT. A document matches a query if the set of terms representing the document satisfies the Boolean expression representing the query. In a pure Boolean model retrieved documents are not ranked. Extended Boolean models, such as those employing fuzzy set theory, can be used to rank the results, as discussed by Lee [53]. Moreover, some variations of the Boolean model allow proximity queries which provide a means of specifying that two terms in a query must occur close to one another in a document, where closeness may be measured by bounding the allowed number of intervening words or by reference to a unit such as a sentence or paragraph.

Ranked Models

When the number of search results matching a given query becomes large, the efficiency of the pure Boolean model goes down to undesirable levels. Since the documents are not ranked, the user is forced to search through long lists of results before they find the most relevant document, which can be at the bottom of the list. A way to avoid this is to rank matching documents in decreasing order of estimated relevance [58]. The search process then becomes a two step process. The first step finds an initial set of candidate documents. The second step ranks all matching documents found in the first step, according to their predicted relevance to the query.

The first step can be performed by returning all the documents that contain any of the words in the query. This can be achieved by applying an OR boolean operator on the query terms.

A common way of ranking documents is the *vector space model* proposed by Salton et al. [77]. In the vector space model, queries and documents are treated as vectors from an n-dimensional vector space, where n is the number of distinct terms in the text collection. The similarity between a document and a query is then measured by their cosine distance, that is, the angle between the two vectors in the n-dimensional space:

$$\cos\theta = \frac{v1 \cdot v2}{\| v1 \| \| v2 \|} \tag{2.1}$$

In this view of documents and queries, also known as the *bag of words* model, the exact ordering of the terms in a document is ignored. The vector view only retains information on the number of

occurrences. Looking at the vector space model in its basic form, one is inclined to ask if all terms in a document are equally important. Some terms have little or no discriminating power in determining relevance [58]. For example, a collection of government documents is likely to have the term *government* in almost every document, and thus in that context the term has little discriminating power. Moreover, in every language there are words that do not convey much meaning — for the English language, these include the words *the, is, are, it, they, for,* etc. Therefore, terms are usually assigned weights during ranking.

The *TF.IDF* [48] is the standard term weighting approach used by many information retrieval systems. TF.IDF multiplies two components — the Term Frequency (TF) and the Inverse Document Frequency (IDF) — to obtain a composite weight for each term. The TF of a term t in document d, is the number of occurrences of t in d. The IDF is defined as follows;

$$idf_t = \log \frac{N}{df_t} \tag{2.2}$$

Where N is the total number of documents in the corpus and df_t is the document frequency, which is the total number of documents containing term t. The idf_t of a rare term is higher than that of a common term. The composite term weight, TD.IDF [48] is defined as follows;

$$tf \cdot idf_{t,d} = tf_{t,d} \times idf_t \tag{2.3}$$

Where $tf_{t,d}$ is the term frequency for term t in document d.

Thus the score of a document d is the sum, over all query terms, of the $tf \cdot idf_{t,d}$ weights for each query term appearing in d [48].

$$Score(q,d) = \sum_{t \in q} tf \cdot idf_{t,d} \tag{2.4}$$

Various schemes using the $tf \cdot idf_{t,d}$ term weighting function have been considered to normalise and scale the term weights to address issues that can bias the term weighting function. One such issue is the effect of document length: longer documents, as a result of containing more terms, will have higher $tf_{t,d}$ values.

The Lucene[56] term weighing scheme is as follows:

$$Score(q,d) = \sum_{t} \left(\frac{tf_{q,t} \cdot idf_t}{norm_q} \cdot \frac{tf_{d,t} \cdot idf_t}{norm_{d,t}} \cdot boost_t \right) \cdot \frac{overlap(q,d)}{\mid q \mid} \tag{2.5}$$

Where:

$\frac{tf_{q,t} \cdot idf_t}{norm_q}$ is the length normalised query weight.

$tf_{d,t} \cdot idf_t$ is the $tf \cdot idf$ from the document.

$norm_{d,t}$ is the term normalisation of t in d

$boost_t$ is a user specified term boost.

$\frac{overlap(q,d)}{|q|}$ is the proportion of the query matched.

The majority of information retrieval systems use ranked retrieval models. In addition, information retrieval systems generally support free text queries as apposed to supporting well defined queries that follow a precise logical structure such as relational algebra or SQL. The next few sections discuss how to construct an index that supports ranked retrieval and free text queries.

2.1.3 Index Construction

Performance of search engines depends on the algorithms used but also, to a large degree, on the underlying hardware. Since the collection to be searched is stored on disk, hard disk read/write and random access rates affect searching performance. In comparison to disk capacity, disk sequential read/write and random access speeds have not increased dramatically in recent years [12]. Thus sequentially reading the entire contents of a disk can take several hours depending on the disk capacity. Because of hard disk performance limitations, a search infrastructure which requires an exhaustive scan of a hard disk in order to process a search query is not a feasible solution. In order to realise efficient full-text search, a data structure needs to be built that contains information about all occurrences of search terms within the collection. The aim of building such a data structure is to reduce the amount of data read from disk and the number of random disk access operations performed while processing a query.

Several data structures have been proposed to support full-text retrieval. The most common are inverted indices [43] and signature files [26]. Inverted indices are the most effective structure for supporting full-text retrieval [95].

Inverted Indices

An *inverted index* [43] (also referred to as an *inverted file* or *postings file*) is a structure that maps each term t, in a given text collection, to a *posting list* that identifies the documents that contain that term. An appearance of t in a given document is recorded as a single *posting* in the posting list. Posting lists may be stored either in memory or on disk. For a large collection, all postings cannot fit in the computer's main memory. Instead, the majority of all postings data will typically be stored on disk. For efficiency at search time, each postings list is stored contiguously; if the

list is fragmented, the overhead from disk seek times would lead to unacceptable query response times. In a typical implementation, a posting consists of a document identifier, number of times the term occurs in each document and the position of the term in the document. The set of terms that occur in the collection is known as the *dictionary* (also referred to as the *vocabulary* or *lexicon*).

When a text collection is read from disk, it is easy to obtain all the terms that are contained in a particular document by sequentially reading that document. However, finding all documents that contain a certain term would require scanning all the documents to find if they contain the term in question. The aim of constructing an index is to change this ordering such that it will be straightforward to obtain the information about all the documents that a specific term occurs in.

If a text collection is considered as a sequence of tuples of the form:

(document, term)

Then the job of the indexer is to reorder the tuples such that they are sorted by term instead of by document.

(term, document)

Figure 2.2 shows an example of converting tuples in this way. Once the tuples are in the latter form, constructing a full inverted index (see Figure 2.3), is a straightforward task.

At a conceptual level, the process of creating an inverted index can be summarised as follows [58]:

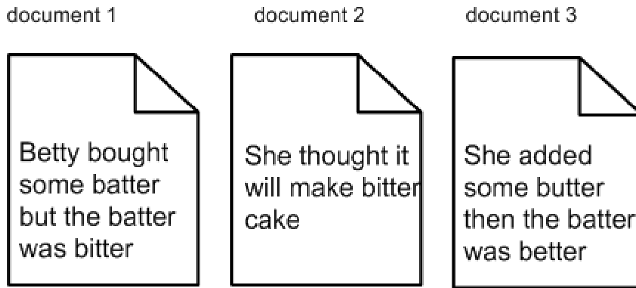
Parsing. The parse step identifies the structure within documents, extracting text in the process. This includes recognising fields such as title, date and the actual document body.

Tokenising. The tokenise step scans documents for terms/tokens which can be words, numbers, special characters and hyphen-separated words, etc. The tokenise step also records positional informational regarding the individual occurrences of words.

Language preprocessing. This includes several preprocessing aspects such as case-folding, stopword removal and stemming. Case-folding converts all terms to a one case, for example lower case. Stopword removal takes out all non-content-bearing words. These are words that do not convey much meaning such as *it, the*, etc. Stemming reduces words to their canonical forms. For example, the use of any of the words *cooking, cooks, cooked* and *cook* should return documents that contain any of these words. The most common algorithm for stemming English words, and one that is implemented by Lucene, is Porter's algorithm [71].

Creating posting lists. This step involves creating the actual inverted index, consisting of a dictionary and posting lists.

Document Collection:



Collection token sequence:

(1, betty), (1, bought), (1, some), (1, batter), (1, but), (1, the), (1, batter), (1, was), (1, bitter), (2, she), (2, thought), (2, it), (2, will), (2, make), (2, bitter), (2, cake), (3, she), (3, added), (3, some), (3, butter), (3, then), (3, the), (3, batter), (3, was), (3, better),

Index token sequence:

(added, 1), (batter, 1), (batter, 1), (batter, 3), (better, 3) (betty, 1), (bitter, 1), (bitter, 2), (bought, 1), (but, 1), (butter, 3), (cake, 2), (it, 2), (make, 2), (she, 2), (she, 3), (some, 1), (some, 3), (the, 1), (the, 3), (then, 3), (thought, 2), (was, 1), (was, 3), (will, 2)

FIG. 2.2: Example of the inverted index construction process.

There are number of ways in which the last step, of creating posting lists, can be carried out. Two such strategies are discussed below.

Sort-based Indexing

Sort-based indexing [12, 58] creates posting lists by first assigning to each term in the vocabulary, a unique term identifier known as a *termID*. Similarly each document has a unique identifier known as a *docID*. Sort-based index construction orders the text collection from tuples of the form (document, term) into tuples of the form:

(termID, docID)

An in-memory data structure is used to translate each term into the matching termID value. Sort-based indexing parses documents into termID-docID pairs and accumulates the pairs in memory until all the memory allocated to the indexing process is occupied. When memory is full, the termID-docID pairs are sorted by termID and all termID-docID pairs with the same termID are collected into a postings list and written to disk as a sub-index of the collection. This is repeated

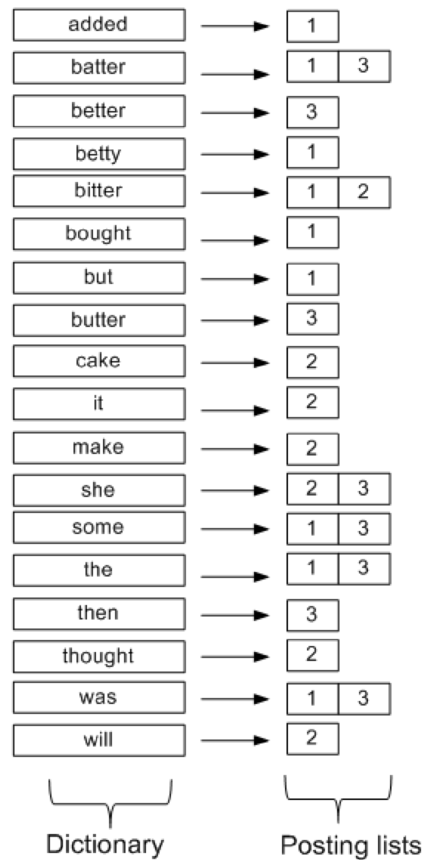


FIG. 2.3: The resulting inverted index. For clarity posting lists only show document identifiers.

until all the documents in the collection have been processed. In the final step all partial indices are simultaneously merged into one large merged index.

Sort-based indexing has several limitations, in particular its excessive disk space requirements. Such space requirements can be cut down by using compression techniques to reduce the amount of disk space consumed by the termID-docID pairs. Better merging mechanisms, for example the one proposed by Moffat and Bell [63], can be used to perform the final merging step using only a constant amount of temporary disk space, reducing space requirements even further. However one shortcoming seems unsolvable — in order to transform each incoming term into a termID, the complete vocabulary of the text collection has to be kept in memory. For a collection comprising many millions of different terms, this can easily exceed the amount of memory that the indexing process is allowed to use.

Merge-based Indexing

Merge-based indexing [12, 46, 58] avoids the large memory requirements of sort-based indexing by removing the need for a data structure that maps terms to termIDs. Merge-based index con-

struction does not use globally unique term identifiers. Instead, each term is treated as its own identifier. This removes the need to maintain an in-memory data structure that translates each term into the matching termID value.

Merge-based index construction orders the text collection from tuples of the form (document, term) into tuples of:

$$\text{(term, document)}$$

An in-memory index is built until memory allocated to the indexing process runs out. When this happens, the entire in-memory index, including the dictionary, is transferred to disk and all memory resources are released. A new dictionary is started for the next chunk of data. The process continues building in-memory indices of data chunks and writing them to disk until the entire collection is processed. In the final step all sub-indices are merged, resulting in the final index for the collection. As in the case of sort-based indexing, the merge method described by Moffat and Bell [63] can be used to eliminate temporary storage space requirements during the final merge step.

Merge-based index construction can be used to build an index using a small amount of main memory because it does not need to maintain a global in-memory term-termID mapping data structure. Moreover, with merge-based index construction it is possible to compress incoming postings on-the-fly, increasing memory efficiency and reducing the number of sub-indices created during the index construction process [12]. Heinz and Zobel [46] claim that their merge-based indexing algorithm is approximately 50% faster than a highly optimised sort-based indexing algorithm. Lucene index construction follows the merge-based index construction approach.

Index Compression

Index compression is a key to efficient evaluation of text queries [95]. Integer compression can be used to reduce query evaluation costs by orders of magnitude for indices stored both on disk and in memory. Integer compression uses a variety of techniques to reduce the magnitude of the numbers stored in postings lists. For example, for each term t in the search engine's vocabulary, an integer field $t : lastPosting$ is maintained, containing the value of the last posting for the term t. Whenever a new posting p has to be added to t's posting list, it is not stored in its raw form, but as a delta value: $\Delta(p) = p - t : lastPosting$. The technique of storing delta values is known as taking *d-gaps*. The delta value can be stored using one of the integer coding schemes. Golomb-coding of d-gaps yields optimal bitwise codes [95]. Alternatively, byte-oriented codes allow much faster decompression [79]. Compression allows more information to be stored in main memory. If the

inverted lists for common terms are cached, responding to queries containing such words reduces to memory lookups, resulting in improved search efficiency.

2.1.4 Index Maintenance

The document collection over which retrieval is provided is either static or dynamic. Static collections such as the Bible or Shakespeare, rarely or never change. With such collections, the index remains the same and it never needs to be updated. However, most data collections frequently change with documents being added, deleted and updated. This means that new posting lists need to be created for new terms and existing posting lists need to be updated. A simple way of achieving dynamic indexing is to periodically reconstruct the index from scratch. However, this solution is only suitable if the number of changes over a period of time is small and a delay in making new documents searchable is acceptable. In order to avoid the complexity of dynamic indexing, some large search engines adopt the reconstruction-from-scratch strategy [58]. At the end of an index reconstruction, query processing is switched from the new index and the old index is deleted. The alternative is to ensure that an existing index structure is synchronised with the changing contents of a text collection by updating the index as changes in the document collection occur.

Index update algorithms follow one general approach — buffering a number of postings in memory and eventually adding them to the existing index in one large physical operation. The strategies that specify the exact details of how the in-memory postings are merged with the existing index can be classified into two main categories: *in-place* and *merge-based* [46, 12].

In-place Index Update

An in-place index update [46, 12] strategy stores each posting list in a contiguous region of the disk and leaves some free space at the end of each list. Whenever postings stored in memory need to be combined with the existing on-disk inverted index, they are appended to the end of the corresponding list. The entire list is relocated to a new place only if there is not enough space left for the new postings. The main limitation of in-place index maintenance is the large number of disk seeks associated with list relocations. Tomasic et al. [92] discuss list allocation policies that determine how much space at the end of an on-disk posting list should be reserved for future updates and under what circumstances a list should be relocated. Shieh and Chung [82] follow a statistical approach to predict the future growth of a posting list based on behaviour observed in the past.

Merge-based Index Update

In contrast to the in-place update strategy, the merge-based update strategy [46, 12] avoids costly disk seeks. When memory allocated to index postings runs out, the in-memory index data is combined with the existing on-disk index in a sequential manner, resulting in a new on-disk index that supersedes the old one. Disk seeks are slower than sequential disk access. Therefore, because all disk operations of the merge-based approach are performed sequentially this is an advantage compared to the disk seeks involved in the in-place update strategy.

Lucene employs the merge-based update strategy. Details of the merge-based index update variant used by Lucene are given in chapter 4.

The index construction methods, the index maintenance methods and the retrieval models discussed so far are applicable to a generic information retrieval system. However, domain specific requirements may need to be taken into account for an information retrieval system to be more effective. The next section is a brief discussion of information retrieval for the Web.

2.1.5 Web Information Retrieval

Web information retrieval has unique challenges and opportunities that are not part of other information retrieval systems. This is due to the unique properties of the Web — its scale, structure, and diversity of backgrounds and motives of its participants [58]. These contribute to making Web search different from searching other collections. Figure 2.4 shows the basic architecture of a typical Web search engine.

Due to the distributed nature of data on the Web, a program called a crawler (also known as a spider) is required to download data from the Web [4]. A crawler typically starts from a set of seed pages, locates new pages by parsing the downloaded pages and extracts the hyperlinks within. The extracted hyperlinks are placed on a queue for further retrieval. Some crawlers use many crawler threads that execute concurrently in order to overlap network operations with CPU processing, thus increasing the throughput — these are called parallel crawlers. The crawler continuously downloads pages from individual servers until local resources, such as storage, are exhausted [4] or the fetch queue gets empty or a satisfactory number of pages are downloaded [1]. The crawler downloads data to the local store which is a snapshot of some part of the Web. The crawler must obtain fresh copies of previously fetched pages to ensure that each indexed pages is a true representation of the Web content. The page revisit policy of a crawler is an important factor in determining the freshness of the indices.

There are two common types of crawlers in literature, batch and incremental crawlers [76, 18]. A batch crawler erases all pages downloaded before and starts from scratch in every indexing cycle. An incremental crawler never erases its local store. It begins where it left off when the last

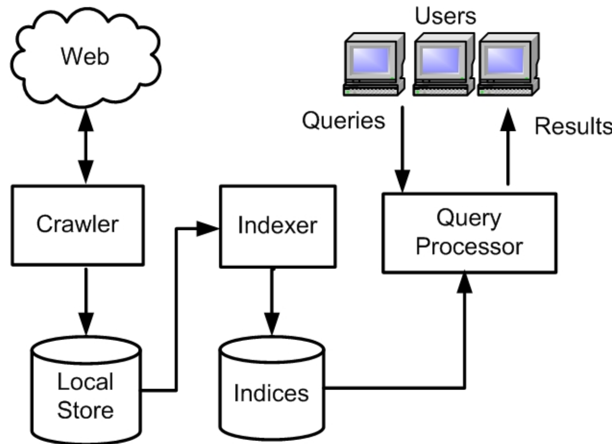


FIG. 2.4: Typical architecture of a Web search engine.

indexing phase stopped [76].

The analysis of hyperlinks and the graph structure of the web is one of the factors considered in ranking documents of a Web search engine. Such link analysis facilitates the computation of a query independent score for Web documents. The query independent score represents the general importance of each document. Two popular link analysis algorithms are the HITS [50] and PageRank [97] algorithms. In addition, link analysis also extracts text that is contained in links pointing to Web pages. Such text is often referred to as *anchor text* [59]. The indexer can leverage the page descriptions embedded into its incoming links by making use of anchor text. For example, the anchor text terms can be included as terms under which to index the target Web page [58].

Thus far, an overview of concepts in information retrieval has been presented. The next section discusses concepts related to grid computing. After presenting relevant concepts in grid computing, a discussion of previous work related to this thesis is given.

2.2 GRID COMPUTING

Technological advances of the Internet and Web have prompted researchers to explore aggregating distributed resources with the aim of solving large scale problems of multi-institutional interest. Their research has evolved into a field in Computer Science, referred to as grid computing [34, 33]. Grid computing is often referred to by several names and given different definitions by many groups of people — global computing, Internet computing and Peer-to-Peer (P2P) computing [13] are all used to refer to grid computing.

The goal of grid computing is often described using the electric power grid as an analogy, where electricity users get consistent, pervasive, dependable and transparent access to electricity without

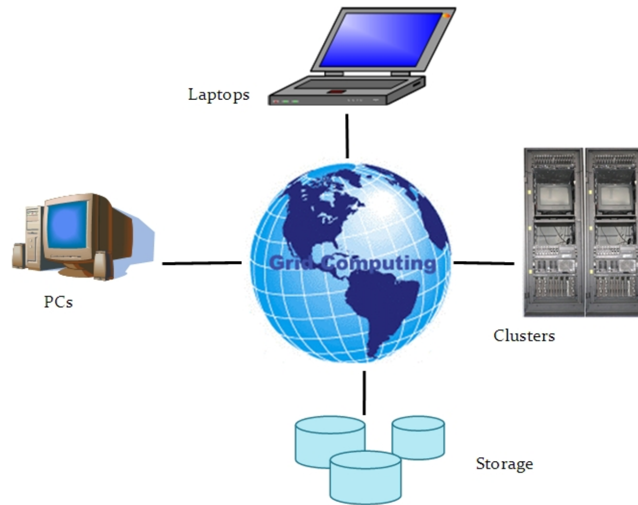


FIG. 2.5: A conceptual view of the grid.

knowing about how it was generated [13]. In the light of this analogy, grid computing has the vision of enabling pervasive computing where users gain access to computing resources including computers, storage systems, data sources and specialised devices, without knowing where those resources are physically located. Figure 2.5 depicts the concept of grid computing.

Computational grids were initially motivated by large-scale, computational and data intensive scientific applications that require more resources than a single computer could provide in a single administrative domain. Grids are developed to allow users to solve larger-scale problems by pooling resources together. The grid infrastructure can benefit many applications, including collaborative engineering, data exploration, distributed supercomputing and service-oriented computing.

Grid computing is related to distributed computing and to parallel processing. As with distributed computing, grids enable physically separated computers to work together on tasks. As with parallel processing, these tasks may be shared in such a way that each computing element can count on particular resources being available to it. Grid computing extends these models by adding end-to-end security, an interoperability model and a standards body [66].

2.2.1 Grid Architecture

The grid architecture depicted in Figure 2.6 is multilayered in nature, where each layer has a specific function [34, 61, 66].

The *fabric layer* provides the actual grid resources, such as computers, storage systems or sensors that are connected to the network to which shared access is mediated by grid protocols.

The *connectivity layer* functionality provides communication and authentication protocols required for grid-specific transactions. Communication protocols allow for the exchange of data between

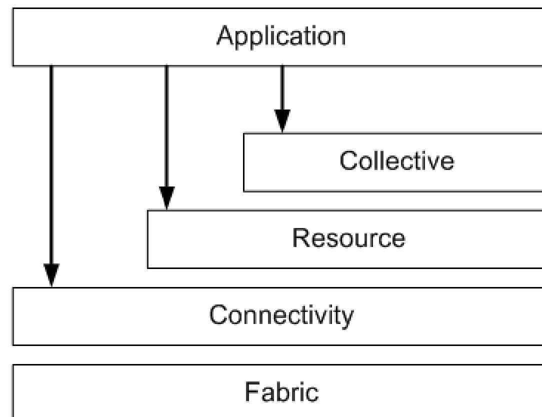


FIG. 2.6: The Grid layered architecture.

fabric layer resources and include transport, routing, and naming protocols such as TCP, UDP and DNS. Authentication protocols provide secure mechanisms for verifying the identity of users and resources, and must rely on existing standards such as X.509, TLS and SSL.

The *resource layer*, also referred to as the middleware layer, defines protocols, APIs, and SDKs for secure negotiation, initiation, monitoring, control, accounting, and payment of sharing operations on individual resources.

The *collective layer* defines protocols, APIs and SDKs that allow interaction across collections of resources. This level provides services that include directory services to discover resources.

At the top of the stack is the *application layer* which includes applications usually developed using Grid-enabled languages and utilities, as well as portals and development toolkits to support the applications. This is the layer that grid users interact with.

Grids can be roughly classified into two categories — global computing grids and institutional grids — each of which is discussed below.

2.2.2 Global Computing Grids

Global computing systems typically harvest the computing power provided by individual computers, using otherwise unused bandwidth and computing cycles in order to run very large and distributed applications [31]. Example systems include FightAIDS@Home [28] and SETI@Home [81]. FightAIDS@Home uses idle computer machines to work on discovering new anti-AIDS drugs. SETI@Home analyses radio telescope data from the Arecibo radio observatory for signals that might indicate extraterrestrial intelligence.

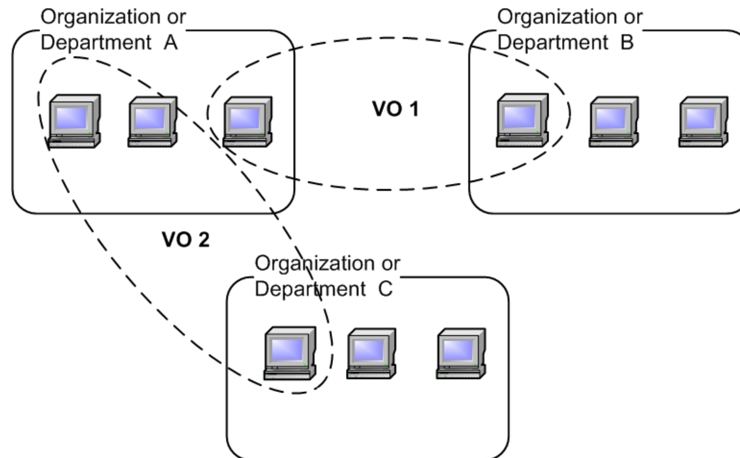


FIG. 2.7: Virtual Organisations enable organisations or departments to share resources.

2.2.3 Institutional Grids

Institutional grids are based on the idea of virtual organisations (VO) [34, 33]. A VO is a dynamic collection of resources and users unified by a common goal and potentially spanning multiple administrative or organisational domains. Depending on context, VOs can be small or large, short-lived or long-lived, single-institutional or multi-institutional, and homogeneous or heterogeneous. Figure 2.7 depicts the notion of a VO.

Institutional grids are implemented via a multipurpose standards-based service infrastructure. There are standardisation efforts in the institutional grid community with the aim of enabling interoperability among different institutional grid deployments. One such effort is the Open Grid Services Architecture (OGSA) [35]. OGSA defines a standard service-oriented institutional grid framework by merging Web services with grids. The goal of OGSA is to standardise all the services in an institutional grid, such as VO user management, security, resource management, job management and data services. The implementation of these services provides a visualisation of a grid's resources and form an institutional grid middleware. However, global computing systems use specialised protocols to provide functionality specific to the system. Thus there is no interoperability among current global computing communities.

2.2.4 Global Computing vs Institutional Grids

Institutional grids are made of established communities with a certain degree of trust for one another. These communities work toward a common goal of creating and maintaining a grid infrastructure. Within such communities one can be held accountable for inappropriate behaviour. In contrast, global computing enforces anonymity and participants have no common goal and thus do not act cooperatively [31]. Furthermore, most of the resources in a global computing

environment are home computers. Global computing obtains computing power by running jobs on desktop workstations only when the keyboard and CPU are idle. If a job is running on a workstation when the user returns and hits a key, thats job may stop or get migrated to a different workstation depending on the scheduling policies of the global computing application in question. On the other hand, institutional grids typically involve more powerful, more diverse and better connected resources such as clusters, storage systems and databases that are administered in an organised manner. Such resources are used to deliver non-trivial qualities of service. Institutional grids usually involve thousands of computing resources — global computing communities have scaled to several million resources.

Both global computing and institutional grids are usually built using grid middleware systems. Grid middleware is an important part in developing robust and useful grid environments. Grid middleware with increasing user bases include the Globus Toolkit [38, 32], the Storage Resource Broker (SRB) [10, 73] and Condor [20, 55]. The Globus Toolkit is the de facto standard for grid middleware and it is discussed in more detail in the next section. The Globus toolkit is not used in the system developed in this thesis — it is discussed here because it is the reference implementation that illustrates what a typical grid middleware system provides.

2.2.5 The Globus Toolkit

The Globus Toolkit (GT) [30, 38] is developed by the Globus project which is a research effort that seeks to develop open source software and its associated documentation for grids. The Globus Toolkit provides protocols and services for computational grids. It is made up of components that implement core grid services such as security, resource location, resource management, data management, resource reservation, and communications. Such services enable grid application developers to select services that meet their specific needs. The Globus Toolkit is a practical implementation of the standards defined by the Open Grid Services Architecture (OGSA) [35]. OGSA adopts and extends Web Services semantics, interface definition mechanisms and conventions originally aimed at the development of distributed business applications using the Internet. Because the Globus Toolkit is built around standard protocols and services, it can be seamlessly integrated with existing networks.

The Globus Toolkit is based on a layered architecture in which higher-level services can be developed using the lower level core services [32]. It emphasises hierarchical integration of grid components and their services. This feature encourages the usage of one or more lower level services in developing higher-level services.

The services provided by the Globus Toolkit include resource-level functionality. For example, it has enquiry software which allows discovering structure and state information for common

resource types such as computers (e.g., OS version, hardware configuration), storage systems (e.g., available space) and networks (e.g., current and predicted future load) [34].

In terms of security, the Globus Toolkit includes a public-key based Grid Security Infrastructure (GSI) [32] protocol which can be used for authentication, communication protection and authorisation. GSI builds on and extends the Transport Layer Security (TLS) protocols to include single sign-on, delegation, and integration with local security solutions such as Kerberos.

The Globus Toolkit also provides resource and status information which it provides via a Lightweight Directory Access Protocol (LDAP)-based network directory called Metacomputing Directory Services (MDS) [29]. MDS consists of two components: the Grid Index Information Service (GIIS) and the Grid Resource Information Service (GRIS). GRIS implements a uniform interface for querying resource providers in a Grid for their current configuration, capabilities, and status. Higher-level tools such as resource brokers can perform resource discovery by querying MDS using LDAP protocols.

Moreover, the Globus Toolkit provides job scheduling components. However, it does not supply scheduling policies, relying instead on higher-level schedulers. It also incorporates an HTTP-based Grid Resource Access and Management (GRAM) [21] protocol used for allocation of computational resources and for monitoring and control of computation on those resources. The GridFTP [3] data management component of the Globus Toolkit extends the File Transfer Protocol (FTP) to include use of GSI security protocols, partial file access, and management of parallelism for high-speed transfers.

With its many services, the Globus toolkit is undoubtedly a rich grid middleware toolkit, however it is not used in this thesis mainly because the grid used is based on a single organisation and thus it does not require most of the services provided by the toolkit. A local scheduler and cycle scavenging technology, such as Condor, and data grid middleware, such as SRB, were deemed adequate for the work of this thesis.

2.3 RELATED WORK

2.3.1 Intranet Search

The search requirements of users within an intranet are different from those of Internet users. One major difference is that users within an intranet look for specific information, for example a definition of a term. This is in contrast to Web search which has broad base of users who in many cases are satisfied with an average result. Li et al. [54] describe Information Desk, a system that uses a “search by type” approach which can be viewed as a simplified version of question answering. Search by type used in Information Desk is based on information extraction where

different information types are extracted from documents. Information types include employee's personal information, homepages of groups and topics and also term definitions. With such a system, users are able to get responses to questions such as "who is", "what is", etc.

A number of commercial systems dedicated to intranet search have been developed. FAST [27], Autonomy [6], OmniFind [68] and other traditional enterprise search engines are software toolkits. They can be installed and customised for the target intranet, however, these toolkits do not mention the hardware infrastructure required to handle large scale intranet search. It is up to the organisation to figure out the hardware infrastructure with the storage and computational power to deliver the desired scalability. The Google Search Appliance and the Google Mini Search Appliance [40] are hardware devices that provide intranet search able to handle up to millions of pages. These devices have initial acquisition costs as opposed to using resources already at the organisations disposal in conjunction with a small number of dedicated resources.

2.3.2 Grid-based Information Retrieval

Grid-based search engines have not been explored as much as cluster-based search engines have. One major development in the area of grid-based search is the GridIR working group [66, 42], which is based on the work of the Open Grid Forum [69]. GridIR seeks to develop standards for information retrieval on computational grids. The GridIR working group proposed a generic architecture for a grid-based information retrieval system. This architecture is made up of three components: the Collection Managers (CMs) which access, store, transform and deliver data items from collections; Indexers that make up the core traditional information search system; and Query Processors (QPs) which can interact with multiple indexers, over time, to gather query responses, merge them and present them to users. Figure 2.8 depicts the generic architecture proposed by the GridIR working group. The GridIR standardisation process is a work in progress. The search engine developed in this thesis follows the general principles proposed by the GridIR working group.

Hybrid Scavenger Grid Architectures

Although there has been work done on information retrieval for cluster, grid or peer-to-peer architectures, there has been virtually no published work that proposes the use of a hybrid scavenger grid for information retrieval. However, a few works have investigated the use of hybrid scavenger grid architectures for other applications. A recent PhD dissertation [5] investigated the use of a combination of dedicated and public resources in service hosting platforms. The study did simulations of workloads of a data stream processing application. The study observed that by designing appropriate resource management policies, the two combined types of resources can be utilised to

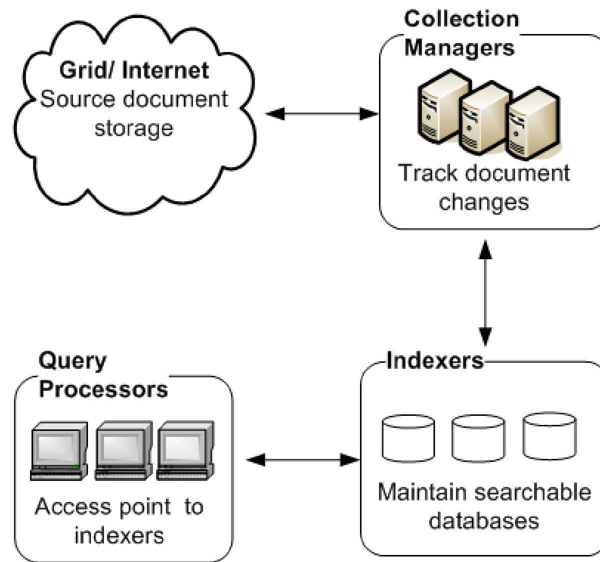


FIG. 2.8: The GridIR proposed architecture.

increase the overall resource utilisation and throughput of the system.

Kenyon et al. [49] performed mathematical analysis and argue that commercially valuable quality assured services can be generated from harvested public computing resources, if some small amount of dedicated computers can be augmented with them. Using mathematical models of harvested cycles, their work has measured the number of dedicated resources necessary to achieve a level of quality assurance.

The peer-to-peer video streaming platform BitTorrent [72] relies on unused uplink bandwidth of end-user computers. Das et al. [22] have proposed the use of dedicated streaming servers along with BitTorrent, to provide streaming services with commercially valuable quality assurances while maintaining the scalability of the BitTorrent platform. With analytical models of BitTorrent and dedicated content servers they have demonstrated how guaranteed download time can be achieved through augmentation of these platforms.

Other Grid Architectures

Most of the previous work related to grid-based information retrieval undertook experiments in large scale institutional grids and their primary focus is on the resource sharing aspect of grid computing. Scholze et al. [45, 80] developed a grid-based search and categorisation tool, GRACE. GRACE was deployed on the European Data Grid, a large distributed network made of dedicated computer clusters. Furthermore, GRACE does not provide immediate results for the search queries — instead results are delivered over time via links sent by email. Cambazoglu et al. [1] describe the architecture of a Search Engine for South-East Europe (SE4SEE). They run their experiments

on a large network, SEE-GRID. The SEE-GRID infrastructure is a large network of computers located in different regions of South-East Europe.

The DILIGENT project [23] aims to provide, among other services, searching and retrieval over grid-based digital library material. The goal of DILIGENT is to generalise the notion of resource sharing created by grid technologies in order to enable the on-demand creation of digital libraries. The focus of DILIGENT is on resource sharing.

Sanderson and Larson [52, 78] discuss the low level details of Cheshire3, a framework for grid-based digital library systems and information retrieval services that operates in both single-processor and distributed computing environments. The focus of Cheshire3 is to provide fast XML search services with identifiable objects and known APIs to handle the retrieval functions needed in distributed digital libraries in a Grid environment.

In their 2006 paper, Meij and de Rijke [60] describe GridLucene, an extension of Lucene [56] that runs on a grid. They investigated how open source retrieval engines can be deployed in a grid environment. They did so by extending Lucene with grid-specific classes. Their work is carried out in an institutional grid environment where they used the tools developed by the European Data Grid project for job submission and SRB [10, 73] as storage middleware. They report results of their preliminary experiments with document collections ranging from 1000 to 10000 XML documents, with an average document size of 50 KB. GridLucene was tested on very small scale collections consisting of a few gigabytes of data as compared to tens and hundreds of gigabytes used in the experiments of this thesis.

2.3.3 Issues in Distributed Information Retrieval

Parallel Indexing and Querying

Distributed information retrieval results in an index that is partitioned across several machines. The index partitioning scheme in a distributed search environment should enable efficient query routing and resolution. Moreover, the index distribution should be balanced across the machines in order to ensure that the system can scale to high query workloads. There are three alternative implementations for distributing the index over a collection of query servers [91, 75, 58, 8]. One alternative is to distribute terms and their corresponding posting lists evenly across all servers — *partitioning by terms* (also known as *global index organisation*). The second alternative is to distribute documents evenly over all servers and an inverted index is built for each server — *partitioning by documents* (also known as *local index organisation*). The last alternative is to use a hybrid index partitioning approach that combines term and document partitioning approaches in one of several possible ways. Although there has been work on hybrid partitioning approaches

[97], document and term partitioning remain the most popular approaches in practice, possibly due to the complex nature of implementing hybrid approaches.

If the chosen approach is partitioning by terms [91], during querying a query is routed to the servers corresponding to its query terms. This allows greater concurrency since queries with different query terms would be handled by different sets of servers. However, multi-word queries require sending posting lists between servers in order to intersect the posting lists of the terms in the query, and the cost of this could outweigh the greater concurrency. Furthermore, it is a challenge to achieve a balanced load with term partitioning. Load balancing terms among the servers is not influenced by a priori analysis of term frequencies, but rather by the distribution of query terms and their co-occurrences. Achieving good partitions is a function of the co-occurrences of query terms and involves clustering of terms [58].

With partitioning by documents [91], each query server is responsible for a disjoint subset of documents in the collection. Thus a search term is broadcasted to all the query servers, each of which returns a disjoint list of relevant documents. Results from the servers are merged before presentation. This type of organisation trades more local disk seeks for less inter-server communication. These disk seeks are performed in parallel and may potentially be faster than the disk access in term partitioning.

The drawback with document partitioning is the difficulty of computing global statistics used in ranking across the entire document collection when the index at any server only contains a subset of the documents. Global statistics can be computed by distributed background processes that periodically refresh indices at each server with fresh global statistics. Document partitioning also requires determining how documents are partitioned among servers [58]. One approach would be to assign all pages from the same host to a single node. The other is to use a hash of each URL into the space of index servers. The problem with the former approach is that on many queries, a large number of results could come from a small number of hosts and hence a small number of index nodes. The latter approach results in a more uniform distribution of query-time computation across nodes [58].

Many large-scale Web search engines use the document partitioning approach. Furthermore, a number [16, 14, 64, 91] of research works have shown document partitioning to be superior when the distribution of terms is skewed, which is often the case in practice. This is because a skewed distribution of terms where some terms are more common than others results in the workload imbalance. The servers with common terms require more time to work on their part of retrieval and the slowest server determines the overall query response time. The experimental system in this thesis implements document partitioning.

Result Merging

The problem of result merging arises from incomparable ranking scores returned by searches of different collections when using the document partitioning approach. The ranking scores are not comparable because each query server computes similarity scores between documents and queries using local collection statistics. Such local statistics are collection dependant and may vary widely across collections. To resolve this problem, various merging strategies can be used. One obvious solution is to use global statistics computed over all the sub-collections. This approach however is not efficient due to the high cost involved in computing global statistics. An approach suggested by Voorhees et a. [93] assumes that each collection contains approximately the same number of relevant items and that they are equally distributed within results lists taken from the collections. Using this assumption, the final result set can then be obtained in a round-robin manner. Evaluations of this approach show that it produces poor results [74].

A different approach normalises the document scores by assigning weighs to each collection. The underlying idea is to use weights in order to increase document scores from collections having scores greater than the average score, and to decrease those from any collections having scores less than the average score. The resulting weight for each collection is then used to modify the score attached to each document. Instead of using document scores directly, each document score is multiplied by the weight of the corresponding collection and the results are merged according to these new scores. Several variants of this solution differ in the way the weights are calculated. Callan et al. [15] consider each collection as a single document. Ranking the collections is similar to document ranking methods used in conventional information retrieval system. Computing the weights in this manner requires maintaining global statistics of the collections. Rasolofo et al. [74] describe a simpler strategy which only uses document scores and result lengths. The collection score is computed according to the proportion of documents retrieved (result length) by each query server. This score is based on the intuition that a collection would contain more relevant documents for a given query if its query server found more documents. Results reported by Rasolofo et al. [74] show that computing the score this way performances competitively with other methods. The search engine developed in this thesis employs this strategy to re-rank the results.

2.4 SUMMARY

Providing fast access to information through information retrieval systems is one of the ways of tackling information overload. Such information retrieval systems need to be based on cost-effective and scalable architectures that can keep up with increase in both data and number of

users.

Enterprise search solutions such as FAST and others are software toolkits and their focus is not on the hardware that powers the search but on the search algorithms. The Google Search Appliance and its variants are hardware devices that can scale to millions of pages, however, they require investing in one or more of the devices.

Constructing an inverted index is a compute intensive job. For this reason, parallel computing architectures such as cluster and grid computing have been investigated as a way of achieving scalable index construction. Grid computing is an attractive alternative when compared to other parallel computing paradigms, as resources within a grid can be viewed as carrying little or no financial cost. Some previous works, such as GridLucene and GRACE, have used institutional grids in the form of large distributed networks of several computer clusters. Although such grids have the advantage that the availability of resources is more predictable than for the resources within a scavenger grid, these grids are not cost-effective as they require investing in the clusters that are the building blocks of such grids. Hybrid scavenger grid architectures have been proposed for other applications but their performance has not been investigated for information retrieval operations.

The next chapter is a discussion of the grid middleware used in this thesis.

CHAPTER

3

Condor and SRB Grid Middleware

In order to construct a robust grid platform, it should be based on robust building blocks. Several grid middleware systems have been developed to enable building grid environments based on functional software which has been tested. The grid middleware chosen depends on the specific requirements of the grid in question. This thesis uses two grid middleware systems, one for workload management — Condor, and the other for storage management — the Storage Resource Broker (SRB). A description of how Condor works and how it was configured for the purpose of the grid used in this thesis, is provided in section 3.1. An overview of how SRB works and the specific configurations used in this thesis are presented in section 3.2.

3.1 CONDOR

3.1.1 Overview

Condor [20, 55] is a high-throughput distributed workload management system for compute-intensive jobs. The goal of a high-throughput computing environment [11] is to provide fault tolerant computational power over long periods of time by effectively utilising computing resources available to the network. Condor can be seen as a batch system because it provides services provided by traditional batch systems, including a job queuing mechanism, scheduling policy, priority scheme, resource monitoring and resource management [88, 85].

Condor works by placing user jobs on a queue and determining when and where to run them based upon a set of rules. It monitors job progress and informs the job submitters upon completion. Condor has mechanisms that enable it to harness wasted CPU power from otherwise idle desktop machines. This presents the ability to use resources whenever they are available, without requiring

one hundred percent availability. This is a feature that the experimental search engine developed in this thesis requires. Condor can run jobs on desktop workstations only when the keyboard and CPU are idle. If a job is running on a workstation when the user returns and hits a key, Condor can migrate the job to a different workstation and resume the job where it left off. The Condor system respects the computer owner's rights and only allocates their resources to the `Condor pool` [1] as indicated by usage conditions defined by resource owners [55].

Condor's remote call capabilities enables it to preserve the job's originating machine environment on the execution machine, even if the originating and execution machines do not share a common file system or a user ID scheme. Furthermore, Condor jobs are automatically check-pointed and migrated between workstations as needed to ensure eventual completion.

Condor has a technique called flocking which allows multiple Condor pools to cooperate, enabling jobs to run in any of the pools with available resources. Flocking allows jobs to run across pools within the same domain. A domain refers to an administrative domain such as separate department or organisation. Condor also enables jobs to run across different domains by means of Condor-G which allows submission of jobs to Grid-enabled resources using Grid Resource Access and Management (GRAM) services from the Globus Toolkit.

Each machine in the Condor pool runs a resource agent which periodically advertises its services to the collector. The collector also receives advertisements from customer agents about their requests for resources. The Condor matchmaker discovers resources via the collector and uses the information to determine compatible resource offers (machines) and requests (jobs). Compatible resource agents and customer agents are notified and if they are satisfied, the customer agents initiate the job in the resource.

3.1.2 Condor Setup

In the experimental search engine developed in this thesis, the role of Condor is to match jobs with available machines, which it does by using its `ClassAd` [88] mechanism. The ClassAd is a representation of characteristics and constraints of both machines and jobs. A Condor pool consists of a single machine which is the central manager and a number of other machines. The central manager performs matchmaking and collects information for the pool. In the grid used in this thesis, a single machine serves as the central manager and the dynamic and dedicated nodes are the machines of the pool which run jobs.

There are a number of steps involved in preparing to run a job in a Condor pool: The first step is to prepare the job to run in batch mode such that it can run unattended; the second step is to select one of Condor's seven runtime environment; the third step is to write a submit description

[1] A complete Condor system consisting of jobs and machines is called a Condor pool.

file which contains information about the job and the last step is to submit the job to Condor. There are number of daemons [20] involved in a Condor system which are involved in tasks from job matching to running the actual job. Descriptions of each of the daemons are given below.

condor_master is responsible for keeping all other daemons running on each machine in the pool.

condor_startd represents a machine in a Condor pool by advertising a machine ClassAd containing attributes about the machine's capabilities and policies. When a `condor_startd` is ready to execute a job it spawns the `condor_starter`.

condor_starter spawns the remote job on a machine and it also prepares the execution environment and monitors the job once it is running.

condor_schedd represents the jobs to the Condor pool and all machines that allow users to submit jobs need to run the `condor_schedd` daemon.

condor_shadow runs on the machine where the job was submitted whenever the job is executing. It handles requests for file transfers, logs the job's progress and reports statistics when the job completes.

condor_collector collects all the information about the status of the Condor pool. The rest of the daemons send ClassAd updates to the collector. These ClassAds contain information about the state of the daemons, the resources or the resource requests they represent.

condor_negotiator performs all the matchmaking within the Condor pool.

As shown in Figure 3.1, in the Condor setup used in this project the central manager can only submit jobs but it does not run jobs. Every other machine in the pool can submit and run jobs. To add a job to the Condor pool, the `condor_submit` command line tool is used. It reads a job description file, creates a ClassAd, and gives that ClassAd to the `condor_schedd`. This triggers a negotiation cycle during which the `condor_negotiator` queries the `condor_collector` to discover available machines and jobs that need to be executed. The `condor_negotiator` then performs matchmaking to match jobs with machines.

Once a match has been made, the `condor_schedd` claims the appropriate machine. When the `condor_schedd` starts a job, it spawns a `condor_shadow` process on the submit machine and the `condor_startd` spawns a `condor_starter` process on the corresponding execute machine. The shadow transfers any data files required to the starter, which spawns the user application. When the job is complete or aborted, the starter removes every process spawned by the user job and frees temporary disk space used by the job.

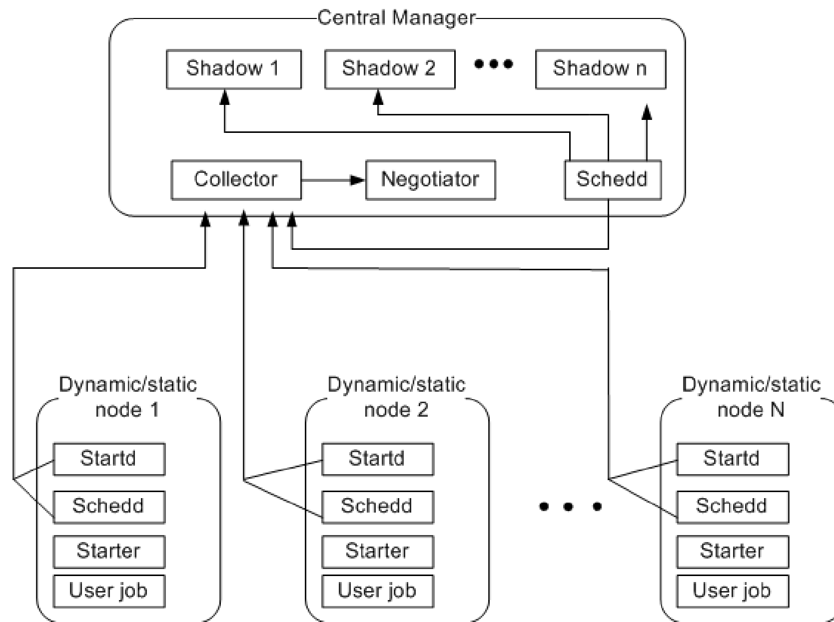


FIG. 3.1: Daemon layout as deployed in the grid used in this thesis. In this case the grid is busy and all the nodes are running jobs.

3.2 THE STORAGE RESOURCE BROKER

3.2.1 Overview

The Storage Resource Broker (SRB) is client-server middleware that virtualises data space by providing a unified view to multiple heterogeneous storage resources over a network [73]. Data virtualisation allows distributed files to appear to be local and can be controlled, organised, and manipulated as if they were on the local disk. SRB is essentially software that is in-between users and resources and it provides a uniform API to heterogeneous storage resources in a distributed fashion. Figure 3.2 is an illustration of how SRB provides storage abstraction.

As shown in Figure 3.2, SRB can provide access to data stored on heterogenous storage resources ranging from archival resources such as HPSS to file systems such as the Unix File System and Mac OSX File System to databases such as Oracle, DB2 and Sybase. SRB provides an abstraction layer over the actual storage devices. This abstraction allows data items which are stored in a single SRB collection to be stored on heterogeneous and geographically distant storage systems [10]. Furthermore, SRB provides capabilities to store replicas of data, for authenticating users, controlling access to documents and collections, and auditing accesses.

The available interfaces include: a UNIX-like file I/O interface; a Java API called JARGON [83]; and a command line-based interface, called Scommands [73], that supports get and put opera-

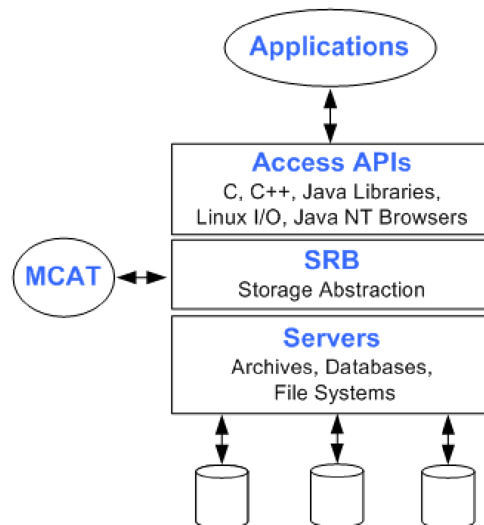


FIG. 3.2: SRB storage abstraction.

tions of individual files, directories or entire collections. SRB also provides a metadata catalog (MCAT) [73] service, by which collections can be annotated, queried and retrieved, providing an abstract definition for resources. These annotations take the form of user-definable attribute-value pairs. They can be arbitrarily modified and queried, for example through JARGON [83]. The metadata stored on MCAT is used for searching at the semantic level and discovery of relevant data objects searching for attributes with which they are annotated. In this way MCAT provides location transparency by enabling access to data sets and resources based on their attributes rather than their names or physical locations.

The design of an SRB server is composed of two separate servers, SRB Master and SRB Server [83]. The SRB Master is the main daemon listening continuously on a well-known port for connection requests from clients. Once a connection from a client is established and authenticated, it `forks` and `execs` a copy of the SRB Server, which is called an SRB agent, to service the connection. From that point onward, the client and the SRB agent communicate using a different port and the SRB Master goes back to listening for more connections. A client can use the same SRB agent to service multiple requests.

SRB is a federated server system in that a group of SRB servers coordinating with each other to service client requests can be configured to form a federation. This enables location transparency which allows any SRB server to access data from any other SRB server. It also enables fault tolerance resulting from the ability to access data using global persistent identifiers — the system automatically redirects access to a replica on a different storage system when the first storage system is unavailable.

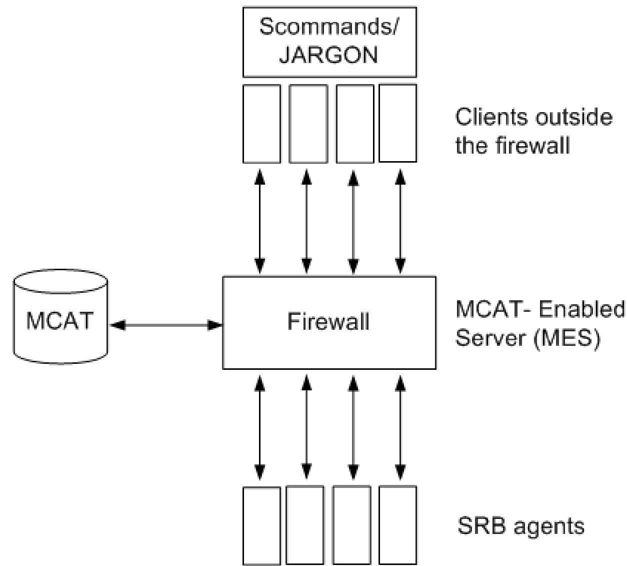


FIG. 3.3: SRB setup in the grid.

3.2.2 SRB Setup

An SRB setup is made up of three logic layers: the clients, the metadata catalog (MCAT) and the SRB servers. Clients query the MCAT to find distributed data objects, replicate, transfer or synchronise data, and many other functions. One of the SRB servers hosts the MCAT database — this server is known as the `MCAT-Enabled Server (MES)`. The rest of the servers are referred to as SRB agents or SRB servers.

In order for the clients to interact with SRB, they need to connect to any of the SRB servers. Connections are required to the MES for MCAT queries and to SRB agents for data transfers. In the grid environment used in this thesis, the SRB agents are within a cluster which is behind a firewall. Any access to these nodes has to be made via the firewall, which is run by the Scheduler. SRB does not explicitly cater for firewalls. In this case, there are a number of possible options that can facilitate going around the firewall. One option is to have the firewall node host the MES, which is able to connect to both external and internal nodes (with reference to SRB agents within the cluster). The problem with this option is that it forces all communications through the MES and it requires disabling the parallel data transfer feature of SRB. An alternative option is to have the firewall host the MES while allowing parallel data transfers but force connections to be initiated by the SRB agents inside the firewall. The latter option was favoured as it does not compromise the fast data transfer feature provided by SRB. Figure 3.3 shows the SRB data grid setup on the dedicated nodes of the architecture.

The clients outside the firewall are the dynamic nodes which request data transfer to and from SRB agents. The clients either use the Scommands or the JARGON Java API to interact with SRB. SRB

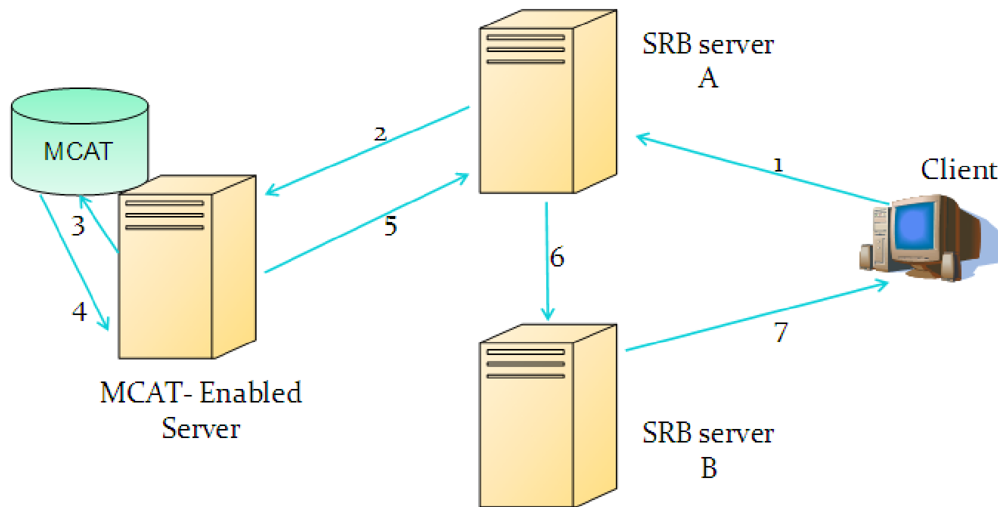


FIG. 3.4: Steps involved in retrieving data from SRB.

agents run on the dedicated nodes. Downloading a file from SRB involves the following sequence of events (see Figure 3.4):

1. The client sends a request for a file to one of the SRB agents, agent A
2. The SRB agent A contacts the MCAT Enabled Server (MES).
3. The MES translates the query into SQL and sends the query to the database hosting MCAT.
4. The MCAT returns the SQL query response to the MES
5. Details about the location of the file are returned to SRB agent A.
6. SRB agent A contacts SRB agent B which is hosting the file requested by the client
7. SRB agent B starts transferring the data to the client.

3.3 SUMMARY

Condor and SRB, the grid middleware systems used in this thesis provide workload management and storage management, respectively. Condor works by matching jobs to machines. In the setup used in this thesis, a single machine acts as the matchmaker whereas all the dedicated and dynamic nodes are candidate machines that can run jobs. SRB provides a layer of abstraction over

heterogenous resources to expose a uniform view of distributed data to all nodes in the grid. The SRB setup used in this thesis has SRB agents behind a firewall. To enable clients outside the firewall to interact with the SRB agents, interactions were limited to server-initiated interactions in order to also ensure that the parallel I/O capabilities of SRB could still be used. Details of how Condor and SRB work as part of the search engine follow in the next chapter.

CHAPTER

4

Design and Implementation

The basic question this thesis attempts to answer is whether it is beneficial to use a hybrid scavenger grid for search engine indexing and searching. With this question in mind, an experimental search engine was designed and implemented, and deployed on a hybrid scavenger grid. This chapter presents the design and implementation of the developed search engine. The discussion explains how the tools used integrate into the system as a whole and what new components were developed.

4.1 SEARCH ENGINE ARCHITECTURE

The architecture of the experimental search engine has five main components (see Figure 4.1). The `User Interface` provides an access point through which queries enter the system. The `Scheduler` performs job allocation and uses `Condor` to distribute jobs to the dedicated and dynamic nodes which run the `Worker Agents`. Worker Agents refer to the software that executes on the nodes of the grid. Dedicated nodes provide persistent storage, which is uniformly accessed via `SRB` storage middleware.

The Scheduler has the task of splitting the data collection into chunks and ingesting the chunks into SRB. The Scheduler also starts the Worker Agent software on the dedicated and dynamic nodes. It does this by first contacting Condor to get a list of the available nodes. The Scheduler then creates Condor jobs that instruct the nodes to run the Worker Agent software. Worker Agents request data to index. Upon receiving a request for a data chunk, the Scheduler allocates a new chunk to the requesting machine. The Scheduler specifies the data chunk allocated to the machine by indicating the location of the chunk on SRB. The Worker Agents run a Lucene indexer on the chunk and ingests the resulting sub-index on SRB. Once all the chunks are indexed, all the sub-

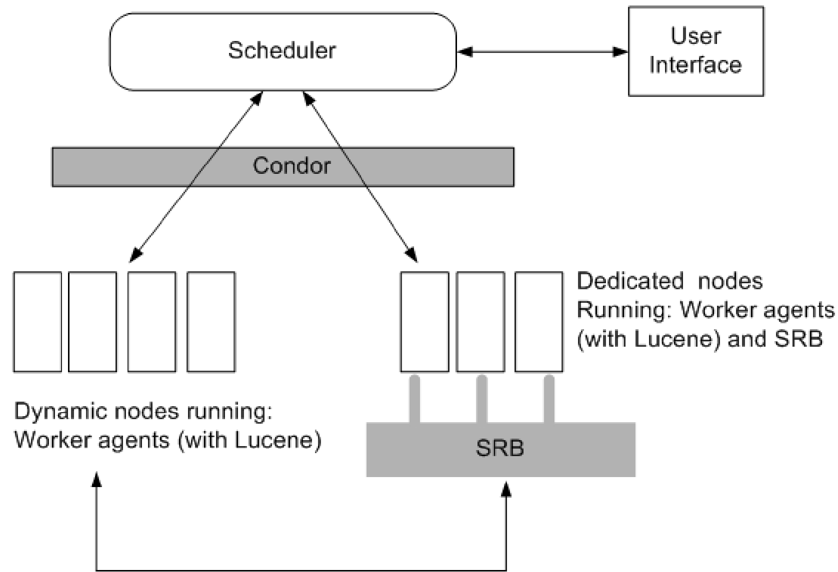


FIG. 4.1: High level components of the experimental search engine architecture. These are: User Interface, Scheduler, Condor, Worker Agents and SRB.

indices located on a single SRB server are merged into a single index.

When a query is posed to the system via the User Interface, it is passed on to the Scheduler which routes the query to all the SRB storage servers that store indices. The SRB servers independently search their indices and return their results to the Scheduler. Finally, the Scheduler merges the results before returning them to the user.

The details of the design of each of the main components are presented in later sections of this chapter.

4.2 INTER-COMPONENT COMMUNICATION

In order for the various components to communicate successfully, the experimental system uses defined communication protocols between the components. This makes it possible to evolve each of the components independently of one another. Some of the components communicate via Web services and others via plain sockets, as explained below.

4.2.1 User Interface to Scheduler Communication

The Scheduler software includes a query handling component which accepts user queries. This component is implemented as a Web service — it exposes a set of operations which can be invoked by clients. Clients can communicate with the server using XML messages that follow the SOAP communication protocol. A SOAP engine is used to interpret SOAP requests and to create SOAP

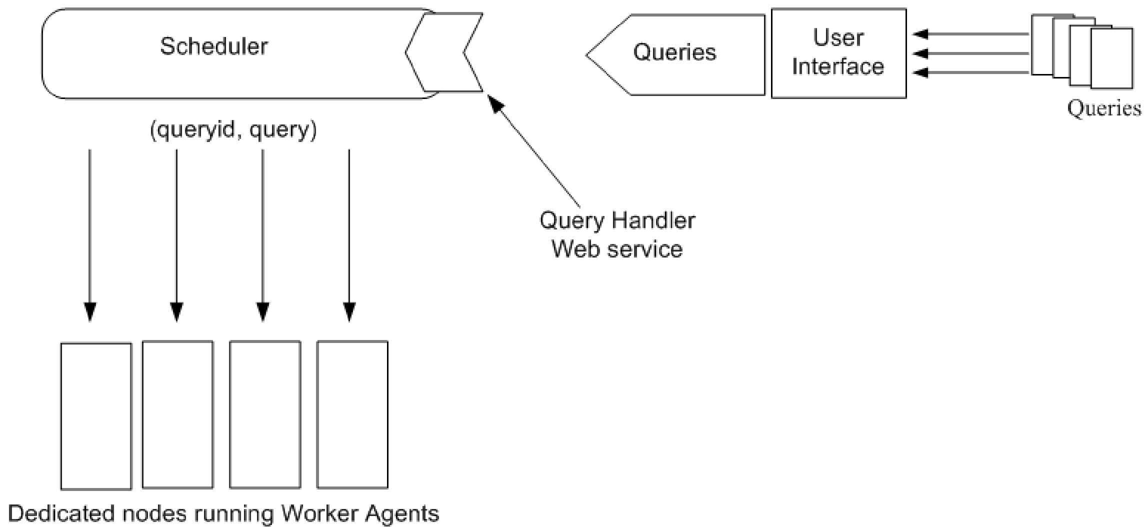


FIG. 4.2: The Query Handler part of the Scheduler is exposed as a Web service. The Scheduler passes queries on to the Worker Agents via socket connections.

responses. This thesis uses the Apache Axis [90] SOAP engine and the execution environment for the Web service is provided by the Jakarta Tomcat [89] application server.

Communication between the User Interface and the Scheduler begins when the User Interface invokes the query handling Web service. The User Interface passes the query as a string, to the query handling method exposed by the Web service (see Figure 4.2). The query handling method returns a list of results ordered in decreasing order of estimated relevance.

4.2.2 Scheduler to Worker Agents Communication

On arrival at the Scheduler, a query is sent to all the Worker Agents running on the dedicated nodes which hold partial indices. The Scheduler opens socket connections to all the Worker Agents and sends the query along with a unique query identifier (see Figure 4.2). The query identifier is attached to the partial results that are returned by the Worker Agents to the Scheduler for merging.

For both indexing and querying, when the Scheduler starts the Worker Agents, the necessary files need to be transferred. These are the executable files that will run on the node and other required `jar` files. Since not all the Worker Agent nodes share a file system with the Scheduler node, these files are transferred using the Condor file transfer mechanism.

During indexing, once the Worker Agents are running on the nodes, they start requesting chunks to index by establishing a socket connection with the Scheduler.

SRB allows uniform access to data stored across the grid and thus the Scheduler only needs to provide the Worker Agents with the name of the SRB directory to index, without specifying details

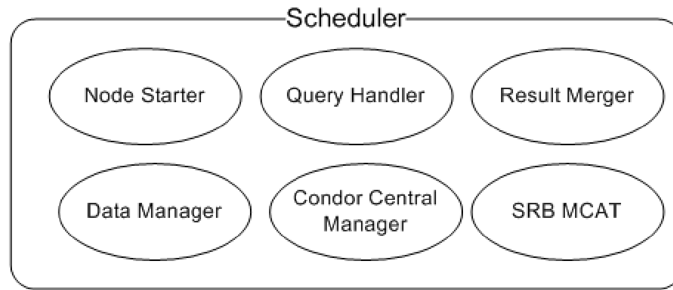


FIG. 4.3: Components of the Scheduler.

of the physical storage resource on which the data is stored. The Worker Agents retrieve the data from SRB using the Java API to SRB. The transfer occurs by means of SRB I/O capabilities which are faster than FTP and scp when transferring large files because of the SRB's parallel I/O capabilities (multiple threads each sending a data stream on the network). For small files, transfers can be a slower due to the additional interaction with the MCAT metadata catalog [83].

4.3 THE USER INTERFACE

The user interface is a lightweight process that provides an entry point for queries to the search engine. The user interface is text-based and it issues queries and waits for responses. A query consists of words or phrases such as "grid search engine" or "data scalability". A query response consists of a list of document identifiers ranked by values which indicate the probability that the document satisfies the information need represented by the query. The user interface presents query results such that the highest ranked documents are at the top.

4.4 THE SCHEDULER

The Scheduler (see Figure 4.3) is the central component of the architecture. Correct operation of the search engine requires that the constituent components of the Scheduler run successfully. The components are: the Condor Central Manager, which is the Condor component that manages jobs and machines; the SRB MCAT which is the SRB metadata catalog; the Node Starter which starts the Worker Agents on the nodes; the Data Manager which splits data into chunks and also allocates chunks to indexers; the Query Handler which accepts queries into the search engine; and the Result merger. The Condor Central Manager for matching jobs to machines and the SRB MCAT for storing SRB metadata have been discussed in chapter 3. The Data Manager, the Query Handler and the Result Merger are discussed in this section.

4.4.1 Node Starter

The Node Starter is responsible for starting the Worker Agent software on the nodes. It does so by scheduling Condor jobs to run on the nodes. If the task is indexing, the Node Starter contacts Condor to obtain a list of the available nodes. The Node Starter then creates a Condor job for each of the nodes. The Condor job description file specifies the executable file that the machines should run and also any other files that the executable file requires to run successfully. In the case of indexing the transferred files include the indexing program with all the *jar* files it requires. Figure 4.4 is an example of a Condor job description file to start Worker Agents on 3 nodes for the task of indexing. The specified nodes start the Worker Agents by running the executable `indexjob.JobHandler`.

If the task is querying, the Node Starter reads a file which specifies the names of the storage servers that hold the indices. It then creates Condor jobs to run on the specified storage servers. The transferred files include the querying program and the *jar* files it requires. Figure 4.5 is an example of a Condor job description file to start a Worker Agent on a node for the querying task.

4.4.2 Data Manager

The Data Manager has two tasks: (a) to split data into chunks and ingest it into SRB and (b) to allocate data chunks to Worker Agents during indexing.

Given a collection of size X, it is split into chunks of a user specified size. In this project chunks of 500 MB were used. The problem with larger chunks is that they take longer to download from SRB, depending on the speed of the connection. Moreover, larger chunks lead to more lost work in case of failure. If, for, example, a node crashes while it was in the middle of transferring an index of a 2GB chunk, when the work gets reallocated to another node it will take longer to index a 2GB data chunk, than to index a 500MB chunk. When using the Java execution environment Condor provides no means of check-pointing and migration of jobs in case of node failures. Thus the work of a failed node is all lost.

Data splitting involves creating chunks from the collection and evenly distributing the chunks among the SRB storage servers (dedicated nodes). The chunks are ingested into SRB via SRB's command-line-based interface.

The data allocation part keeps track of chunks which have been indexed and chunks which are still to be indexed. The Worker Agents of the dynamic and dedicated nodes make chunk requests via socket connections. The Worker Agents index the data and ingest the resulting sub-indices into SRB. Upon ingesting a sub-index into SRB, the Worker Agent notifies the Data Manager which marks the data chunk in question as indexed and the Worker Agent is assigned another chunk. For a Worker Agent running on a dedicated node (and thus an SRB server), ingesting the sub-index

```
Universe = java
Executable = /home/nnakasho/apps/searchengine/indexjob/JohHandler.class
Log = indexjob.log.$(Cluster)
Output = indexjob.out.$(Cluster).$(Process)
Error = indexjob.error.$(Cluster)
Should_transfer_files = YES
WhenToTransferOutput = ON_EXIT
transfer_input_files = /home/nnakasho/.srb/.MdasEnv,/home/nnakasho/.srb/
.MdasAuth
jar_files = /home/nnakasho/apps/lib/jargon_v1.4.20.jar,/home/nnakasho/
apps/lib/lucene-core-2.1.0.jar,/home/nnakasho/apps/searchengine/indexjob/
requiredfiles/requiredfiles.jar,/home/nnakasho/apps/lib/
jargon_v1.4.20.jar,/home/nnakasho/apps/lib/fontbox.jar,/home/nnakasho/
apps/lib/PDFBox.jar,/home/nnakasho/apps/lib/tm-extractors.jar,/home/
nnakasho/SRB/SRB3_4_2/matrix/lib/activation.jar,/home/nnakasho/apps/lib/
tar.jar,/home/nnakasho/apps/lib/bcprov-jdk15-138.jar,/home/nnakasho/apps/
lib/bcmail-jdk15-138.jar
notification = never
owner = "nnakasho"
Arguments = indexjob.JobHandler
Requirements = machine == "pc05.tsl.uct.ac.za"
Queue
Arguments = indexjob.JobHandler
Requirements = machine == "pc06.tsl.uct.ac.za"
Queue
Arguments = indexjob.JobHandler
Requirements = machine == "pc16.tsl.uct.ac.za"
Queue
```

FIG. 4.4: Example Condor job description file to start Worker Agents on 3 dynamic nodes for the task of indexing

into SRB is a simple task. However, for a Worker Agent running on a dynamic node ingesting the sub-index is a multi-step process due to the firewall, which separates the dynamic nodes from the SRB servers as discussed in chapter 3. In such a case, when the Worker Agent finishes indexing a data chunk, it notifies the Data Manager. The Data Manager selects one of the dedicated nodes, in a round-robin manner, to handle the sub-index ingestion. The selected node first downloads the data from the remote dynamic node using the `scp`[1] function. Once the sub-index is on the local disk of the dedicated node, the dedicated node ingests it into SRB and removes it from its local disk.

The Scheduler keeps track of the machines working on indexing jobs. It periodically checks if all the machines are still up. If a machine is found to be unresponsive, the work allocated to it is reallocated to another machine. All the work done by an unresponsive machine is then lost.

[1] scp is a means of securely transferring files between a local and a remote host or between two remote hosts, using the Secure Shell (SSH) protocol.

```
Universe = java
Executable = /home/nnakasho/apps/searchengine/queryjob/JohHandler.class
Log = queryjob.log.$(Cluster)
Output = queryjob.out.$(Cluster).$(Process)
Error = queryjob.error.$(Cluster)
Should_transfer_files = YES
WhenToTransferOutput = ON_EXIT
transfer_input_files = /home/nnakasho/.srb/.MdasEnv,/home/nnakasho/.srb/
.MdasAuth
jar_files = /home/nnakasho/apps/lib/jargon_v1.4.20.jar,/home/nnakasho/
apps/lib/lucene-core-2.1.0.jar,/home/nnakasho/apps/searchengine/indexjob/
requiredfiles/requiredfiles.jar,/home/nnakasho/apps/lib/
jargon_v1.4.20.jar,/home/nnakasho/apps/lib/fontbox.jar,/home/nnakasho/
apps/lib/PDFBox.jar,/home/nnakasho/apps/lib/tm-extractors.jar,/home/
nnakasho/SRB/SRB3_4_2/matrix/lib/activation.jar,/home/nnakasho/apps/lib/
tar.jar,/home/nnakasho/apps/lib/bcprov-jdk15-138.jar,/home/nnakasho/apps/
lib/bcmail-jdk15-138.jar
notification = never
owner = "nnakasho"
Arguments = queryjob.JobHandler
Requirements = machine == "machine1.cluster.aim"
Queue
Arguments = queryjob.JobHandler
Requirements = machine == "machine2.cluster.aim"
Queue
```

FIG. 4.5: Example Condor description file to start Worker Agents on 2 storage servers (dedicated nodes) for the task of querying

4.4.3 The Query Handler

The Query Handler component is the part of the Scheduler which interacts with the user interface. The method exposed to the User Interface takes a string query as its single parameter. The Query Handler routes each query to each storage server holding an index. It also invokes the Result Merger and returns a list of results to the User Interface.

4.4.4 Result Merger

The Result Merger component combines partial result lists sharing the same queryid into a single list and returns the result list to the Query Handler. As soon as the Result Merger has received all partial results relative to a query, it combines the results into a single list. The document scores in the partial results obtained from the different storage servers are not immediately comparable. This is because each storage server computes similarity scores between documents and queries using local collection statistics. Such local statistics are collection dependant and may vary widely across collections. This situation can be rectified by assigning new scores to each document and

re-ranking the documents based on their new scores. One way of doing this is to assign weights to each collection [74]. The underlying idea is to use weights in order to increase document scores from collections having scores greater than the average score, and to decrease those from any collections having scores less than the average score. The resulting weight for each collection is then used to modify the score attached to each document. Instead of using document scores directly each document score is multiplied by the weight of the corresponding collection and the results are merged according to these new scores.

Several strategies for computing collection scores have been proposed. The Result Merger uses a strategy proposed by Rasolofo [74] to calculate collection scores. The collection score is computed according to the proportion of documents retrieved (result length) by each storage server. Calculating the score this way is based on the intuition that a collection would contain more relevant documents for a given query if its storage server found more documents. The score for the ith collection is determined by:

$$s_i = log\left(1 + \frac{l_i \cdot K}{\left(\sum_{j=1}^{C} l_j\right)}\right) \tag{4.1}$$

Where K is a constant, l_i is the number of documents retrieved by the ith collection, C is the number of collections. Based on this collection score, the collection weight denoted by w_i is calculated for the ith collection as follows:

$$w_i = \left[(s_i - \bar{s})\ /\bar{s}\right] \tag{4.2}$$

Where s_i is the score of the ith collection and $\bar{s}$ is the mean collection score.

4.5 Roles of the Nodes

The characteristics of the nodes dictate the type of jobs they can perform. The availability of dedicated nodes is more likely than that of dynamic nodes. The dedicated nodes are dedicated to search engine operations and thus, as long as they are up and running, they are available to execute jobs and to provide services that are required for the search engine to operate. For this reason, the dedicated nodes are responsible for providing persistent storage for the indices and also for responding to queries — these are tasks that require high resource availability. If required, dedicated nodes also can be used for indexing. Dynamic nodes can be re-claimed at any point by their owners and their availability is not guaranteed. Thus dynamic nodes can only be used for jobs that do not require high resource availability. On the other hand, dynamic nodes are available in large numbers. When one is claimed by its owner or it fails, it can easily be replaced by another. Because there are many of them, dynamic nodes can provide scalability for

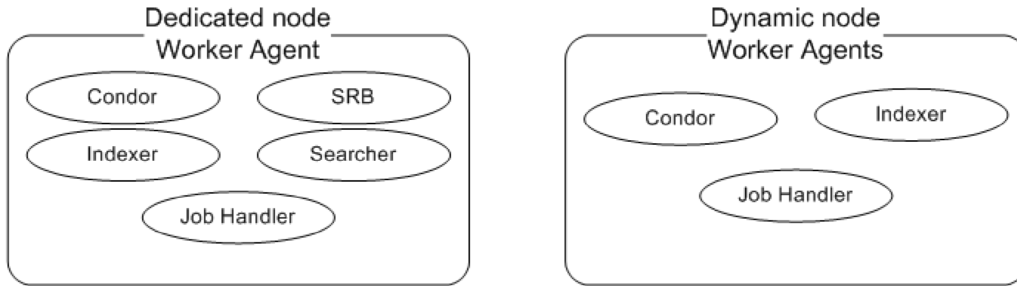


FIG. 4.6: Components of the dedicated and dynamic node Worker Agents.

computationally intensive jobs. Dynamic nodes are therefore suitable to perform text indexing — a computationally intensive task which does not require high availability.

Figure 4.6 shows the components for the Worker Agents for both type of nodes. Dedicated node Worker Agents run the Condor, SRB, Indexer, Searcher and Job Handler components. Dynamic nodes run the Condor, Indexer and Job Handler components. The Indexer and Searcher components are for indexing and querying respectively. The Job Handler on both dynamic and dedicated nodes is the software that runs when a job is allocated to a node using Condor. If the job specified is indexing, the Job Handler has the task of interacting with the Scheduler to request for data chunks to index. Upon receiving a data chunk to index, the Job Handler starts the Indexer and indexes the data chunk. When the node finishes indexing the chunk, its Job Handler requests another chunk. If the job specified is querying, the Job Hander waits for query requests.

4.6 SEARCH ENGINE JOBS

The Node Starter component of the Scheduler has the task of starting the Worker Agent software on the nodes. The Worker Agents are initiated by the Job Handler component of the nodes. The Job Handler accepts work allocations from the Scheduler and uses other components of the node to perform the actual job. This section discusses the details of the jobs performed during indexing and querying.

4.6.1 Indexing Job

The focus of the work of this research is not developing new information retrieval algorithms. Therefore an existing information retrieval engine (Lucene [56]) was used in developing the Indexer and Searcher components.

Core indexing operation are performed by the Lucene information retrieval library. Lucene can index and make searchable any data that can be converted to textual format. This means that Lucene can index and search data stored in different file formats as long as it can be converted

to text. In this project Lucene is used to index HTML, PDF, MS Word, RTF and TXT file formats. Text is extracted from HTML documents by using an HTML parser which also comes with Lucene. PDF documents are parsed using the PDFBox [70] open-source library. MS Word documents are parsed using TextMining.org [87] tools. RTF documents are parsed using classes from Java's standard distribution. These classes are part of the `javax.swing.text` and `java.swing.text.rtf` packages.

Lucene is only a part of the distributed search engine system that was developed. In order to realise a functional distributed search engine, several other aspects need to be implemented. Distributed indexing can be organised in one of two ways:

Local Data indexing. In this type of distributed indexing organisation, machines index data that is stored on their local disks. Upon completing indexing, an index machine transfers the partial index to one of the index SRB storage servers.

Non-local Data indexing. In this type of distributed indexing organisation, indexing machines download source data that is stored on the storage servers on SRB and also store the resulting indices on SRB storage servers.

The type and structure of an organisational intranet will dictate which type of distributed indexing an organisation ends up choosing for its intranet. Intuitively, the Local Data indexing approach achieves superior performance because of data locality and thus it incurs less network transfer time. In organisational intranets where data is naturally distributed, it makes sense to employ the Local Data approach. For example, within an academic institution, desktop machines within the department of Chemistry can be used to index data in that department, whereas desktop machines within the Computer Science department can be used to index data from that department. In other intranets such as corporate intranets, most of the data is likely to be stored on centralised company servers where documents in the form of annual reports and design documents maybe stored. In such cases the Non-local Data approach could be the suitable solution.

If the chosen approach is Non-local Data indexing, then the Indexer performs the following sequence of steps.

1. Download data from SRB
2. Index the data
3. Transfer index to SRB
4. Notify Job Handler of completed chunk

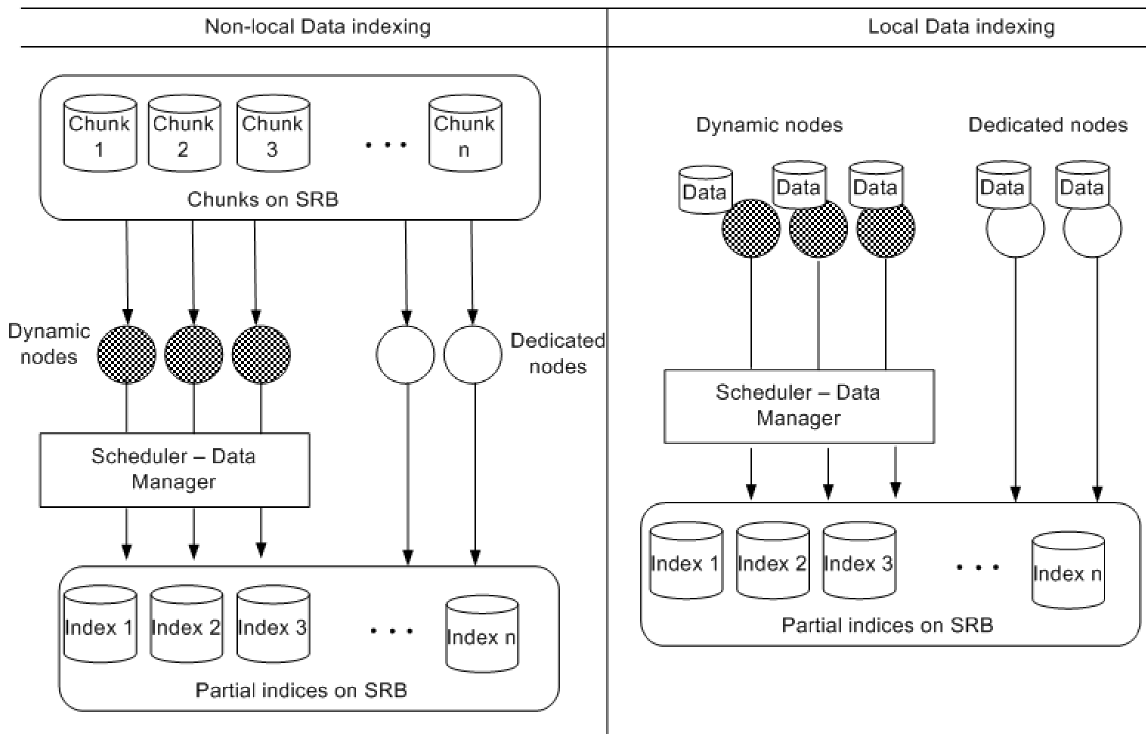


FIG. 4.7: The Non-Local Data and Local Data distributed indexing approaches.

With the Non-local Data indexing approach, the node first downloads the data from SRB before indexing it. The alternative option of indexing directly on SRB by using the file streaming methods of SRB was considered, but it proved to be too slow and error prone.

If the chosen approach is Local Data indexing, then the Indexer only performs steps 2-4 above because the data is already located on the local disks of nodes. Figure 4.7 illustrates the difference between the two approaches.

The index construction process of Lucene uses a merge-based algorithm. An in-memory index is built until memory allocated to the indexing process runs out. When this happens, the entire in-memory index is transferred to disk and all memory resources are released. The process continues building in-memory indices and writing them to disk until the entire chunk is processed. In the final step all sub-indices are merged, resulting in the final index for the chunk.

Index maintenance

When the data collection changes, the index needs to be updated to reflect the changes in the collection. Changes in data collections need to be reflected in the search results in as little time as possible.

Index updates handle the three document update scenarios as follows:

Document deletions. Entries of deleted documents are deleted from the index.

Document modifications. Document modifications are handled in a two step procedure. In the first step, the entry of the old version of the document is deleted. In the second step, an entry for the new version of the document is added to the index. This way of handling document modifications is not necessarily optimal, especially if documents only go through minor modifications.

Document additions. Entries of new documents are added to the index.

The first step of updating the index is to delete stale documents. This involves deleting entries of documents that have been deleted from the collection and also deleting entries of old versions of modified documents. The second step involves adding entries of documents that are new to the collection and also adding entries of new versions of documents that have been modified. Addition of new documents is done in a merge-based index update fashion which is part of Lucene. An in-memory index is built until memory allocated to the index maintenance process runs out. The in-memory index then has to be combined with the old index. Merging the in-memory index with the entire old index every time memory runs out, leads to long index update times. In order to reduce update times, multiple sub-indices are often kept on disk — when memory runs out, the in-memory index is written on disk as a separate independent index [12]. However care has to be taken as to how many index partitions are stored on disk because a large number of partitions leads to fragmented posting lists resulting in slow response times during querying. Several variants of the merge-based update strategy have different ways of controlling the number of sub-indices stored on disk. The update strategy used by Lucene is referred to in literature as `Logarithmic Merge` [12]. Logarithmic Merge imposes a condition on the number of index partitions on disk at any point in time by requiring that each index partition i either be empty or contain a total number of documents between:

$$r^{i-1} \text{ and } (r-1)^{i-1} \tag{4.3}$$

For a collection of N documents, this leads to a total number of indices on disk no larger than

$$log_r(N) \tag{4.4}$$

Logarithmic Merge also leads to fragmented posting lists — on-disk posting lists are not stored in contiguous regions of the disk. This can potentially degrade query processing performance, due to the additional disk seeks that need to be performed to fetch all segments of a query term's posting

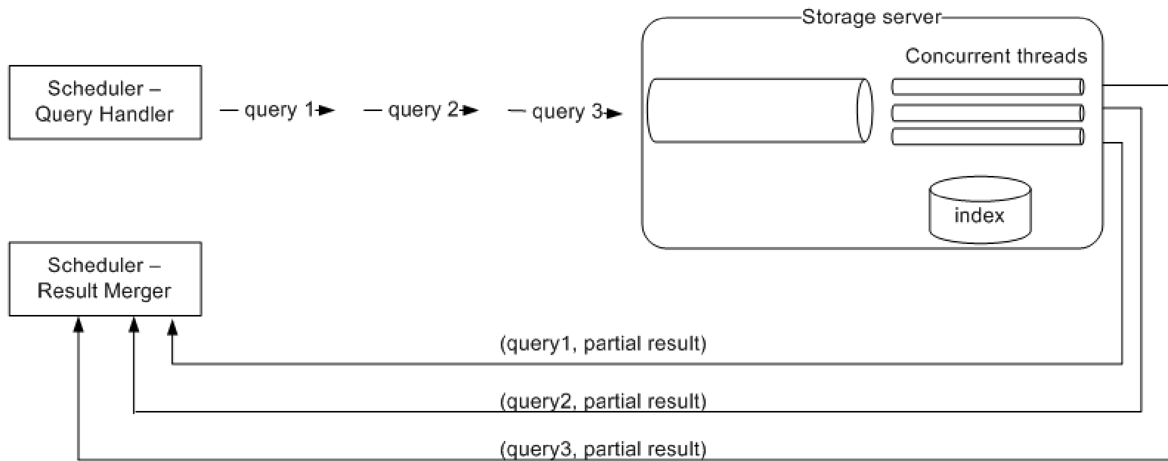


FIG. 4.8: Query processing at a storage server.

list from the individual index partitions. This slowdown, however, is likely to be relatively small, because the number of separate index partitions is tightly controlled.

4.6.2 Querying Job

The index partitioning scheme used in this thesis is document partitioning whereby documents are distributed evenly over all SRB storage servers and an inverted index is built for each server. Therefore, each storage server holds a partial index and processing of queries involves all storage servers.

During indexing, the index resulting from indexing each chunk is written to SRB as a separate index without immediately merging the indices. When all the chunks are indexed, each SRB storage server merges the partial indices that it holds using the index merging methods of Lucene. Merging indices involves concatenating posting lists that belong to the same term into a single posting list, resulting in one single index instead of multiple indices.

When starting a query job on an SRB storage server, the Job Handler reads the index from SRB. As explained already, SRB file streaming methods have proven to be slow and error prone. Therefore to facilitate fast query responses, the storage servers first store the index onto their file systems allowing queries to be subsequently responded to without going through the SRB file system.

To generate a partial ranked answer set, a storage server receives query requests from the Query Handler component of the Scheduler and inserts them in a queue. Queries in this queue are processed concurrently by employing multiple threads (see Figure 4.8).

4.7 SUMMARY

A distributed search engine was developed with the hybrid scavenger grid as the target architecture. It uses Condor for job distribution, SRB for distributed data management and Lucene as the underlying information retrieval engine. Work allocation is managed by the Scheduler. The Scheduler allocates data chunks to the nodes to perform indexing jobs. Indices are stored on SRB storage servers which are dedicated nodes. Query processing starts with the Scheduler routing the query to all the SRB storage severs. Each server generates a partial result set and sends it to the Scheduler. The Scheduler merges and re-ranks the partial result sets before returning a single ranked result set to the user.

The next chapter discusses the evaluation of the search engine.

Chapter

5 Evaluation

This project explores the use of a hybrid scavenger grid for information retrieval operations. The aim is to realise an architecture that is cost-effective yet can scale to medium-to-large search engine workloads. This chapter discusses how the experimental search engine that was designed for a hybrid scavenger grid was tested and evaluated. The evaluation aims to determine the indexing and querying performance of the hybrid scavenger grid, how it utilises resources, and its cost/perfomance ratio. To determine these, a series of experiments were carried out:

1. Grid middleware overhead. The middleware that a distributed system utilises incurs additional overhead in execution time. If such middleware overhead is not well managed, it can overshadow the computing power made available by the distributed resources. Performance of the Grid middleware for distributed storage, including its throughput and impact on indexing, is discussed in section 5.2.

2. Varying grid composition. Having established the performance of the grid middleware and its impact on indexing performance, experiments were then carried out to find out how the hybrid scavenger grid architecture can be best organised in a way that delivers the optimal indexing performance. In particular, tests were carried out to establish the optimal number of dynamic nodes required to index a collection of a given size— in terms of indexing time and resource efficiency. Moreover, tests were carried out to establish the optimal combination of tasks that the dedicated nodes should perform. These tests are presented in section 5.3.

3. Index maintenance. The distributed search engine should not only have the ability to efficiently build an index but also the ability to perform fast index updates. Index updating performance was evaluated, taking into account document additions, modifications and

deletions. Tests related to index maintenance are reported in section 5.4.

4. Hybrid scavenger grid vs multi-core. Presented in section 5.5 is a cost-effectiveness and system efficiency comparison between a hybrid scavenger grid architecture and a multi-core architecture. The aim of this set of tests was to determine the workload conditions under which each architecture is the most beneficial option for search engine indexing.

5. Querying performance analysis. The ultimate goal of a search engine is to respond to quickly queries. Query response times should be similar to what users of Web search engines are accustomed to, which is typically less than a second. Experiments related to query response times are discussed in section 5.6.

Before discussing the experiments listed above, the resources used during the experiments are presented. These include: the data collections used for indexing experiments; the set of queries used for querying experiments; and the hardware resources on which the experiments were run.

5.1 RESOURCES

5.1.1 Data Sets

The data collections used for the experiments are as follows:

- The AC.UK collection.
  This collection is made up of data crawled from the `.ac.uk` domain, which is the domain of academic institutions in the United Kingdom. The reason for crawling data from this particular domain is mainly because it has English text. Although text in other languages is important in cross-language information retrieval, in this case it does not add anything to the collection since the search engine is an English language only search engine. A summary of the collection is in Table 5.1.

- The UCT.AC.ZA collections.
  These collections are two snapshots, taken a month apart, of the `uct.ac.za` domain which is the domain of the institution where the project was carried out. They serve to illustrate index maintenance performance.

The AC.UK collection is the primary one used in most experiments except in places where explicitly stated otherwise. In both data collections, the crawled data was filtered to remove all binary files. The file types retained are HTML, PDF, RTF, TXT and DOC. In order to test for data scalability, the AC.UK collection was duplicated in cases where the data collection needed for the experiments is larger than the actual size of the collection. In particular, data was duplicated in testing indexing

	AC.UK collection	UCT.AC.ZA1 collection	UCT.AC.ZA2 collection
Size of Collection	70.27 GB	10.9 GB	11.6GB
Number of Files	825,547	304,374	326,789
PDF	87,189	8,407	8,493
DOC	21,779	1,862	1,868
TXT	2,569	270	281
RTF	2,042	90	94
HMTL	711,968	293,745	315,881

TABLE 5.1: Summary of the text collections used in the experiments. The AC.UK collection is the result of a crawl of the domain of academic institutions in the UK. The UCT.AC.ZA collections are partial snapshots of the `uct.ac.za` domain taken a month apart.

performance using data sizes greater than 70.27 GB. Query performance experiments use the AC.UK collection as is — the data was not duplicated to simulate larger collections. The reason behind this is that duplicating the data collection only changes the index in one dimension. With duplication, individual posting lists (inverted lists) increase in size but there is no change in the size of the dictionary. This can affect querying performance. It does not, however ,affect indexing performance since in distributed indexing the data is indexed in small jobs and there are no duplicates within each indexing job. Each partial index is independent of subsequent partial indices and thus the index building process is not affected by data duplication.

5.1.2 Query Set

There are two ways to represent queries for experimental purposes. The first option is to use actual queries from query logs. The second option is to generate synthetic queries from probability distributions that are based on actual intranet search characterisation statistics. The first option is more realistic and it allows a comparison between the experimental system and real systems.

Typical query logs from the domains crawled were not available. Instead, test queries used are top queries of the Web search volume as per Google records accessible via the Google Insights for Search service [39]. Google Insights for Search is a Web-based service that provides the most popular queries across specific regions, categories and time frames. Since the collection used for query performance evaluation is the AC.UK collection, the United Kingdom was selected as the region of interest and the time frame selected was “2004–present”. The categories chosen are those that are typically covered by academic institutions. All the queries within the query set return a non-empty result set on the index of the AC.UK collection. A summary of the query set is in Table 5.2 and sample queries are shown in Table 5.3

While it can be argued that the top queries on the Web are not representative of the queries posed within university intranets, an argument can also be made to the contrary. For example, an

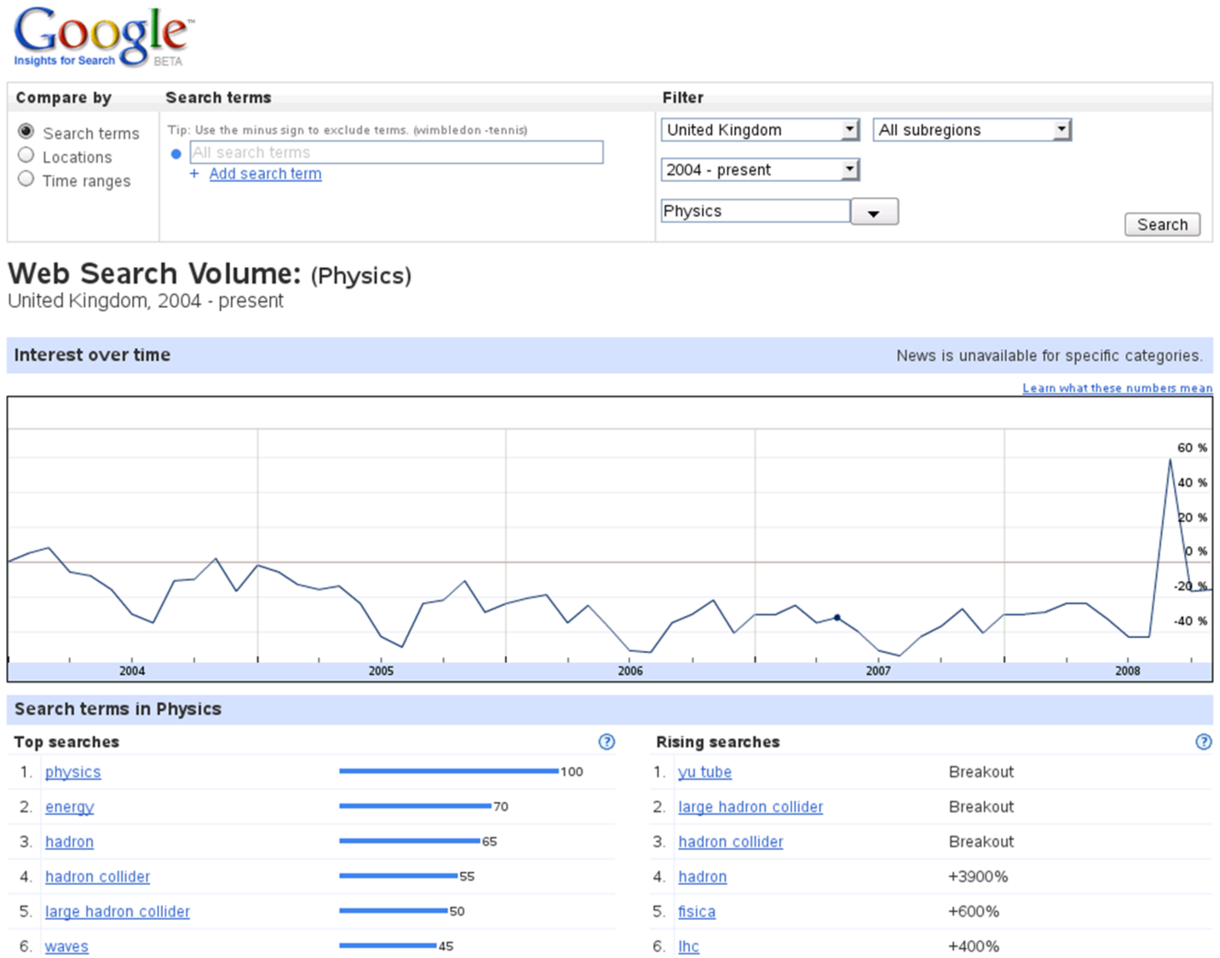


FIG. 5.1: Top searches on Google Insights for Search. The results returned are for the Web search volume of the period 2004–November 2008 for the United kingdom region in the Physics category.

intranet query log analysis by Kruschwitz and Sutcliffe [51] has shown that intranet queries are generally shorter compared to Web search queries. This is a property that intranet queries have in common with top Web queries. For the two sets of query logs used by Kruschwitz and Sutcliffe, the average terms per query are 1.55 for the log consisting of 100 queries and 1.97 for the log consisting of 20,578. The average terms per query for the 1,008 queries extracted from Google Insights for Search is 1.5.

Total number of queries	**Average terms per query**	**Length of longest query**
1,008	1.5	3

Categories
Science
Sports
Telecommunications
Society
Health
Arts and Humanities

TABLE 5.2: Query set summary. Queries are extracted from the Google Insights for Search service.

5.1.3 Hardware and Performance Measurement utilities

The following four computer systems (see Figure 5.2) were used for the experiments.

- A set of machines referred to as dynamic nodes. These are desktop machines within an undergraduate laboratory consisting of about 66 machines. Each machine is equipped with a 3 GHz Pentium 4 processor, 512 MB of RAM and a 40 GB SATA disk drive (7200 rpm). These machines are within a 100 Mbps Ethernet network.
- A set of machines — referred to as dedicated nodes — in a cluster. These are 13 desktop class computers interconnected by a Gigabit Ethernet network. Each machine is equipped with a 3 GHz Pentium 4 processor, 512 MB of RAM and an 80 GB SATA disk drive (7200 rpm).
- A desktop class computer with a 2.33 GHz Intel Core 2 Duo processor, 2 GB of RAM and a 250 GB SATA hard disk (7200 rpm) — the Scheduler — which is used for distributing indexing jobs during indexing as well for distributing queries and merging results during querying.
- A server with a 3GHz Intel Quad-Core Xeon processor, 8 GB of RAM and a 500 GB SATA hard disk (7200 rpm). This is the multi-core machine used in the experiments that compare the hybrid scavenger grid with a multi-core.

The filesystem implementation that was used in all machines is the `ext3` file system as implemented in the Linux 2.6* kernel. The software versions that were used are SRB 3.4.2, Lucene 2.1.0 and Condor 6.8.4. The SRB metadata catalog was stored in a Postgres database, running Postgres 7.4.17. Postgresodbc 07.03 was used for database connectivity.

Aggregate performance statistics, such as the total time to build an index for a collection of size X, were measured by the UNIX `time` command. Finer-grained performance values were obtained

Sample Queries			
age of consent	the solar system	london school economics	tonsillitis
census	telescope	england cricket	throat infection
population	stars	uk post office	british gas
cheerleading uniforms	large hadron collider	youth crime	respiratory system
isaac newton	harvard law school	transmitter	harry potter
uk statistics	carbon emissions	virgin broadband	shakespeare
office national statistics	biodiversity	royal mail delivery	eating disorder
national archives	mouth ulcers	the earth	climate change
world cup cricket	gums	eclipse	mount st helens
digital tv	dental surgery	big bang	olympic games beijing

TABLE 5.3: Sample queries extracted from Google Insights for Search in the United Kingdom region for the time frame "2004–November 2008".

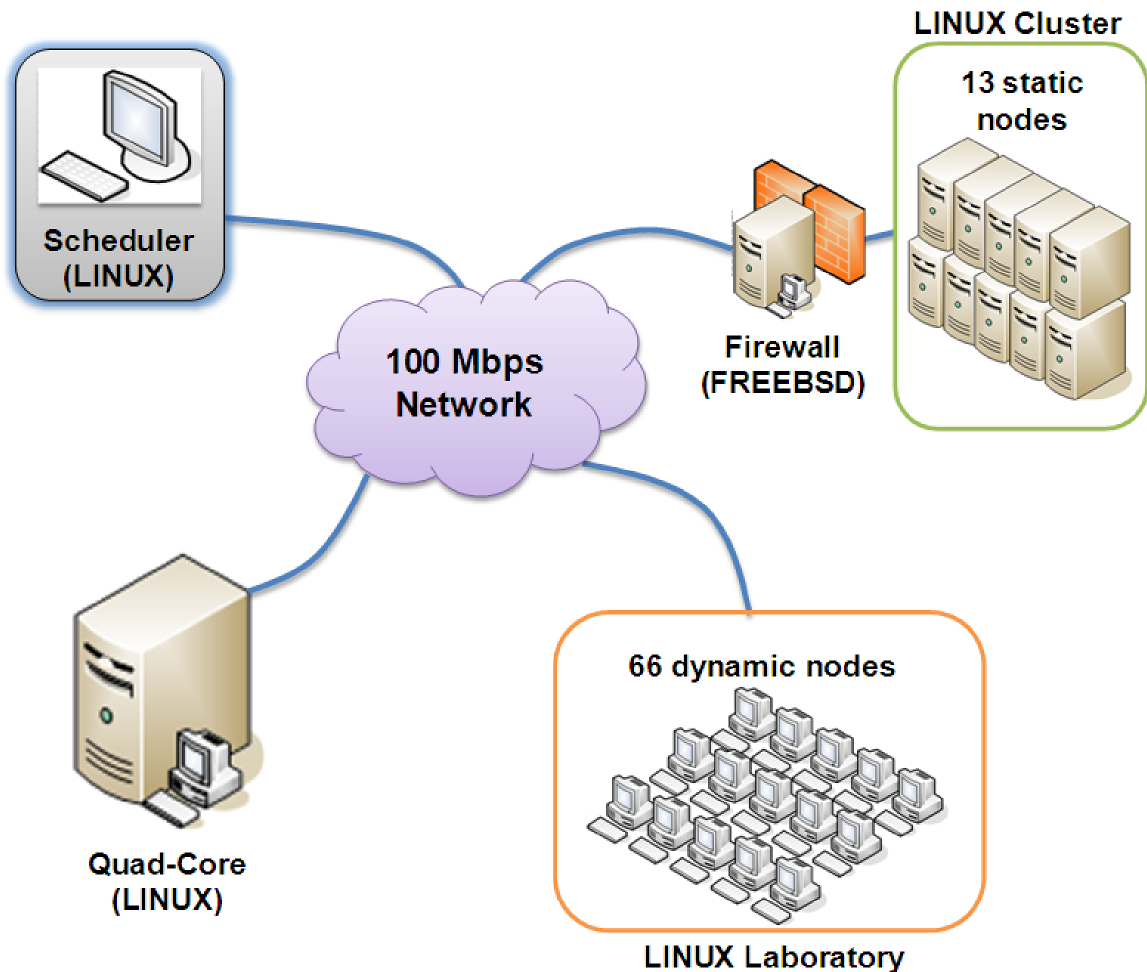


FIG. 5.2: Computer systems used for the experiments. The computer systems are within a 100 Mbps network. The Linux cluster computers are interconnected by a Gigabit Ethernet network.

through Java timing functions, in particular the `Date.getTime()` method.

The first set of experiments evaluated SRB performance. They measured the magnitude of the overhead of SRB. Knowing SRB throughput and its impact on indexing time helps to determine the effect of SRB on results of subsequent experiments.

5.2 GRID MIDDLEWARE OVERHEAD

SRB provides a layer of abstraction over data stored in various distributed storage resources, allowing uniform access to distributed data. Introducing SRB to manage distributed data incurs additional overhead resulting from interaction of SRB servers with the SRB metadata catalog and also from the transfer of data across the network. Ingesting of data into SRB should therefore happen at a fast speed. Furthermore, SRB should utilise the available network bandwidth efficiently. Ideally, the rates of reading and writing to SRB should be close to the maximum available bandwidth. It is clear that these rates will be lower than the available bandwidth because hard disk access speed is another significant factor in SRB operations. Experiments in the next two sections measure SRB throughput and its impact on indexing time.

5.2.1 SRB Throughput

Aim

The aim of this experiment was to determine SRB ingest rates as well as SRB download rates, that is, to establish the read and write speeds of SRB. SRB write speed is the speed of transferring data over the network, registering the data transferred in the metadata catalog and writing the data to the disk of the destination SRB server.

Similarly, the read speed is an aggregate speed, first performing a lookup in the metadata catalog to establish the location of the data, then reading the data from the SRB server that holds the data and transferring the data over the network.

Methodology and Results

To determine the write speed, a file (of size 246 MB — the file size could be any size) was ingested into SRB and the time taken for the operation to complete was measured. The write speed is then the result of dividing the size of the transferred data by the time taken to transfer it. Similarly, the read speed was measured by recording the time to download a file from SRB. Each test was repeated 10 times and the average was recorded.

`Sput` is the SRB command line interface for ingesting data into SRB. `Sget` is the interface for

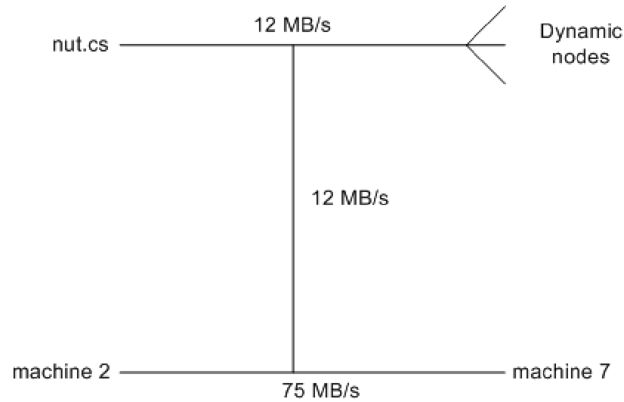


FIG. 5.3: Network connection of the nodes. `machine7` and `machine2` are within a Gigabit Ethernet network with a measured maximum bandwidth of 75 MB/sec. `nut.cs` is connected to machine2 by a 100 Mbps network with a measured maximum bandwidth of 12 MB/sec.

	Source	Destination	Speed
Sput	machine9	machine2	31.07 MB/s
	nut.cs	machine2	8.72 MB/s
Sget	machine2	machine9	31.35 MB/s
	machine2	nut.cs	8.62 MB/s

TABLE 5.4: SRB read and write speeds.

downloading data from SRB. machine2 and machine7 are from the set of dedicated nodes and are within the same network, which has maximum available bandwidth of 75 MB/sec. `nut.cs` is a machine within the same network as the dynamic nodes. `nut.cs` was used in the place of one of the dynamic nodes because the author did not have privileged access required to install an SRB client on the dynamic nodes. The link between `nut.cs` and the dedicated nodes has maximum bandwidth of 12 MB/sec. Figure 5.3 illustrates how the machines are connected. The results of the experiment are shown in Table 5.4.

Discussion

The read and write speeds between machine7 and machine2 are about 42% of the network bandwidth whereas the read/write speeds between `nut.cs` and machine2 are about 72% of the measured bandwidth. Within the faster network connecting machine7 and machine2, SRB read/write speeds are further from the network capacity compared to the slower network whose read/write speeds are closer to network capacity. It appears that within the fast network, the limiting factor is not network bandwidth. Other factors such as disk access and the need to interact with the

database which stores metadata have an impact on the attained I/O speeds. Because bandwidth is a factor in SRB read/write speed, SRB read/write speeds vary with network load. Transfers done during network load peak times will be slower. The transfers in this experiment were not done during peak times.

This experiment revealed SRB throughput in terms of read and write speeds. What is of interest however, is how these speed values affect indexing time. The next experiment measured this impact.

5.2.2 SRB Impact on Indexing

Having established SRB read/write rates, experiments were then carried out to determine its impact on indexing performance.

Aim

The aim of this experiment was to determine if the presence of SRB increases indexing time and, if so, to what extent.

Methodology and Results

The metric of interest is accumulated indexing time. Of particular interest is how much extra time is spent on indexing due to the presence of SRB. As discussed in chapter 4, distributed indexing can be organised in one of two ways. With `Local Data` distributed indexing, machines index data that is stored on their local disks and transfer the partial index to one of the index SRB storage servers. With `Non-local Data` distributed indexing, machines download source data that is stored on the storage servers on SRB and also store the resulting indices on SRB storage servers (see Figure 5.4). Naturally, the Local Data indexing approach achieves superior performance because of data locality and thus it incurs less network transfer time.

The experiment first measured indexing performance of a local Lucene indexer running on a single machine (a dedicated node). The second part of the test measured indexing time of distributed Lucene running on a single indexing machine (both dedicated and dynamic nodes) for both types of distributed indexing. The results of the experiment are in Figure 5.5

Discussion

For the dedicated node cases local Lucene outperforms both forms of distributed indexing as expected. Moreover, the Local Data (dedicated node) distributed indexing approach outperforms the Non-local Data (dedicated node) approach. This is due to the extra network time incurred by

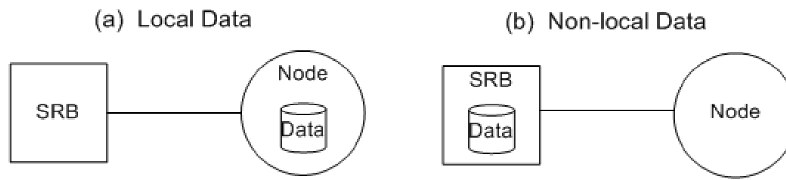


FIG. 5.4: Local Data and Non-local Data indexing

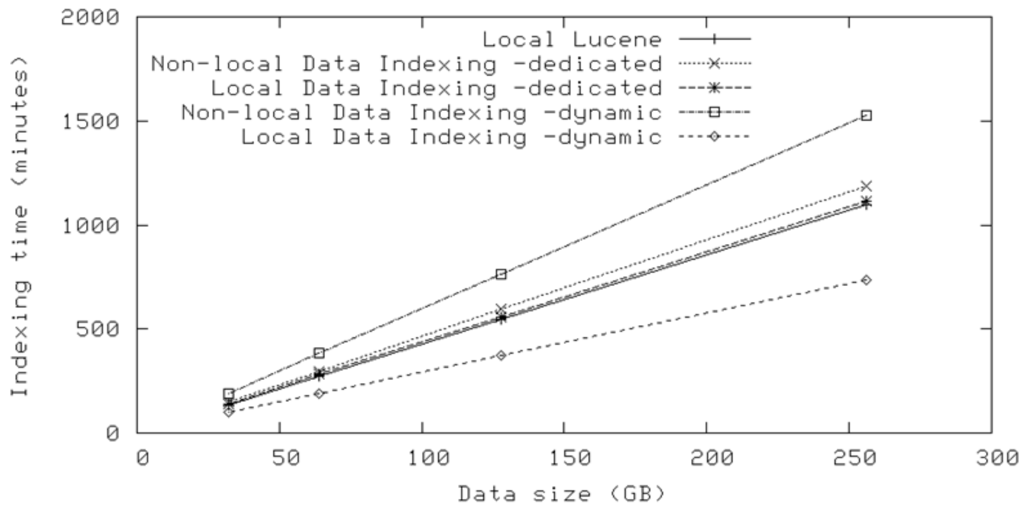


FIG. 5.5: Indexing performance benchmarks. `local Lucene` refers to plain non-distributed Lucene, `dedicated` refers to one of the dedicated nodes and `dynamic` refers to one of the dynamic nodes.

the Non-local Data approach resulting from the need to transfer the data to be indexed. For the dedicated node case, the Local Data approach incurs between 2-4% extra indexing time whereas the Non-local Data approach incurs 8-10% extra time. For the dynamic node case, the Non-local Data approach takes 39-41% more time to index. This is because of the slow network connecting the dynamic nodes to the SRB servers where data is stored, which limits the SRB read and write speeds. The Local Data approach running on a dynamic node performs better than local Lucene running on the dedicated node because the dynamic nodes are faster machines than the dedicated nodes even though their processor speeds and memory are the same. One notable difference between the two types of nodes are the hard disks which are not the same size — seeks times are faster on smaller disks and the dynamic nodes have 40 GB diks while the disks on the static nodes is twice their size. On average, the minimal data chunk takes about 40 seconds longer on the dedicated nodes, which is about 47% more time than the dynamic nodes.

Within a high speed network with maximum measured bandwidth of 75 MB/s, the overhead introduced by SRB is relatively low, with a maximum of 10% extra indexing time. However, in a slow network with maximum measured bandwidth of 12 MB/s, the overhead of transferring data

from SRB is high, with a maximum of 41% extra indexing time.

The limited bandwidth connecting the dynamic nodes to the SRB servers results in long indexing times when using the Non-local Data indexing approach. Therefore, in the subsequent experiments, the Local Data indexing approach was used to evaluate performance of the hybrid scavenger grid, except in cases where it is stated that the Non-Local Data indexing approach was used. Experiments in the next section focused on how the resources of the grid can be best organised and utilised to deliver the best indexing performance.

5.3 Varying Grid Composition

In a non-distributed indexing system, several hardware and software properties affect performance, for example, CPU speed, cache size and hard disk seek times. In a distributed system, additional factors affect indexing time depending on the implementation details of the system. In the hybrid grid-based indexing system developed in this thesis, the following variables contribute to indexing time.

- Number of dynamic indexers. Dynamic indexers are dynamic nodes — these are machines obtained from cycle-stealing.
- Number of storage servers. Storage servers are dedicated nodes — they are responsible for storing the partial indices resulting from completed indexing jobs. They also store the source data when the Non-local Data approach is used.
- Total number of indexers. If only the dynamic indexers take part in the actual indexing of data then the total number of indexers is the same as the number of dynamic nodes. However, if the storage servers perform indexing in addition to storing indices, then the total number of indexers is the sum of the dynamic indexers and the storage servers.

Experiments in this section varied the grid composition in terms of one or more of the above variables as well as in terms of combinations of tasks performed by the nodes.

5.3.1 Varying Dynamic Indexers

Within the hybrid scavenger grid, the number of dynamic indexers plays a major role in indexing time. Ideally, as the number of dynamic indexers increases, indexing time should decrease linearly.

Aim

This experiment aimed to find the number of dynamic indexers that delivers optimal performance. Given a data collection of size X, what is the number of dynamic indexers that realises optimal performance? Optimal in this sense means that indexing time is reduced and also that the indexers involved are well utilised. Utilisation of indexers refers to the amount of work performed by the indexers. Each added indexer incurs additional communication time which adds to the total indexing time. Therefore, indexers need to be kept busy for a reasonable amount of time for their presence to have a substantial impact on performance.

Methodology and Results

Indexing performance for various data sizes was analysed. The accumulated indexing time is the total time spent on all tasks performed to index a collection of a given size, including job scheduling and communication. Indexing time is the time spent on actual indexing of data as opposed to other tasks such as file transfer or job scheduling. Communication and storage time is the time to transfer indices and to ingest the indices into SRB.

The CPU load average was obtained from the UNIX virtual file `/proc/loadavg`. The CPU load average is the average of the number of tasks running or in a runnable state at any instant, as opposed to zombie tasks, tasks waiting on an operation such as I/O, suspended tasks, etc. `/proc/loadavg` shows the average as measured over the last 1 minute, the last 5 minutes, and the last 15 minutes, respectively. In all the experiments the average of the last 5 minutes was used.

Ideally, the load average should stay below the number of CPUs in the machine. Thus on a dual core, it is best if the load average is below 2. When the load average goes over the number of CPUs on the system, this means that there are tasks waiting for CPU time. This may or may not be a problem, depending on what the system is doing and how good the kernel scheduler is.

Performance results obtained from the experiment are shown in Figure 5.6 and Figure 5.7.

Discussion

From Figure 5.6(a) it is clear that as the number of dynamic nodes increases, indexing time decreases. Initially indexing time begins to decrease rapidly, however as more dynamic nodes are added, the rate at which indexing time decreases goes down. This shows a trend of indexing time approaching a minimum indexing time limit. The implication is that, for a collection of size X, there is a number of dynamic nodes after which indexing time does not decrease but remains constant. This number is determined by the size of the collection and the size of indexing chunks as follows:

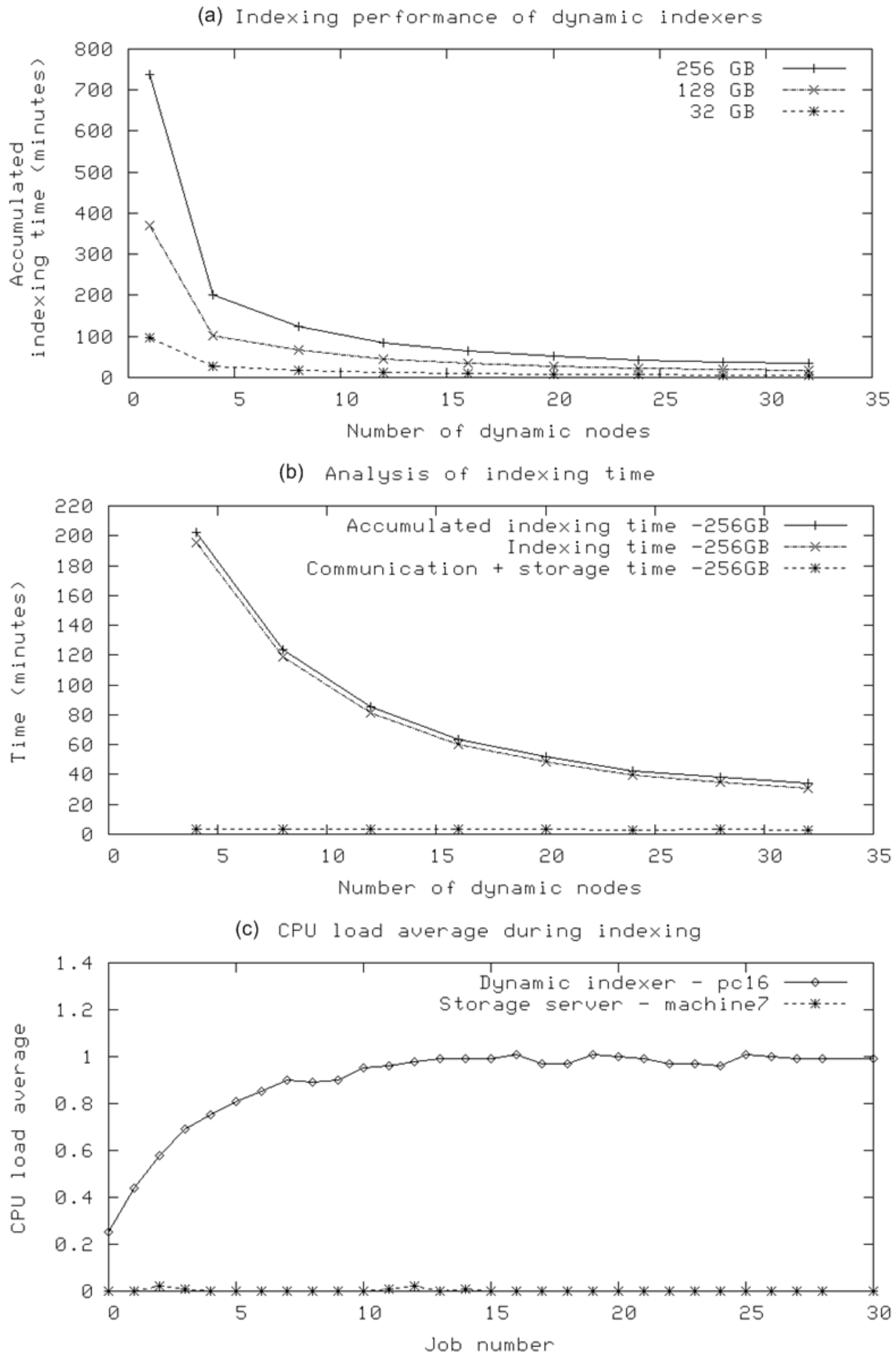


FIG. 5.6: Indexing performance for dynamic indexers. The approach evaluated is Local Data indexing, using 6 SRB storage servers.

Let

$S =$ size of collection	$S_c =$ size of one chunk
$T =$ time to index the collection	$T_c =$ time to index one chunk

N_d = number of dynamic indexers N_c = number of chunks

T_0 = time to index on a single machine

Then

$$N_c = \frac{S}{S_c} \quad , \quad T_c = \frac{T_0}{N_c}$$

$$T = \left\lceil \frac{N_c}{N_d} \right\rceil T_c \tag{5.1}$$

Equation 5.1 shows that the minimum time to index a collection of a given size, S, is T_c — this is obtained when $\frac{N_c}{N_d} = 1$, that is when the number of chunks, N_c is equal to the dynamic indexers N_d. In Equation 5.1, dividing the number of chunks, N_c, by the number of dynamic indexers, N_d, implies that all index servers contribute equally to indexing time. This is not necessarily true. Even though all the dynamic nodes have the same system specifications, some can be slower than others depending on the CPU load average of the machine. Moreover, it is assumed that the sizes of the chunks are equal in terms of MB count. However, the file type composition of different chunks is not the same — a chunk which contains many PDF, MS Word and other types that require parsing into plain text before indexing will take longer to process than a chunk which contains mainly plain text files.

From the component-wise analysis of indexing time shown in Figure 5.6(b) it is clear that a large part of indexing time is spent on actual indexing. Communication and storage time for transmitting and storing indices on storage servers remains more or less the same even as the number of dynamic indexers increases.

Moreover, from Figure 5.6(c) it can be seen that during indexing, the CPU load average on the dynamic indexers increases to a point where it reaches and even slightly goes over the number of CPUs in the machine. The CPU load average on the storage server remains more or less unchanged. This shows that the dynamic indexers are well utilised but also that indexing is computationally intensive and, thus should only be performed on idle desktop workstations as it can noticeably slow down a user's machine. The storage servers on the other hand are under-utilised with their only task being the storage of partial indices. Therefore, the storage servers can be used to do other tasks in addition to storing indices. An investigation into what other tasks the storage servers can perform without overwhelming them is the subject of later experiments.

Figure 5.6 and Equation 5.1 suggest that continued increase in the number of dynamic indexers reduces indexing time up to a minimum. What has not been shown is how resource utilisation is affected as more dynamic nodes are added to the grid. Figure 5.7 shows the grid system efficiency

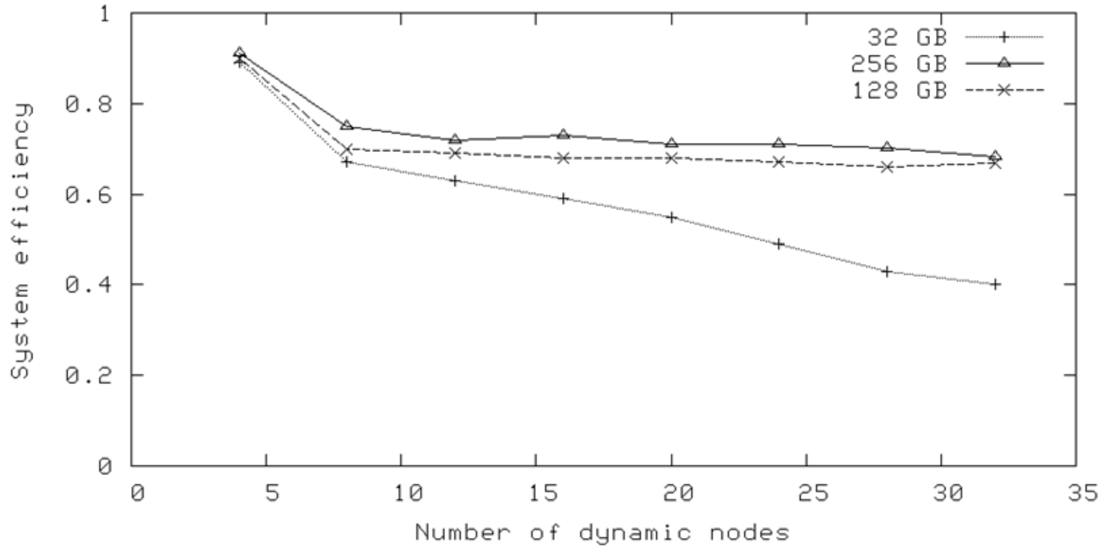


FIG. 5.7: System efficiency during indexing

when varying numbers of dynamic nodes. System efficiency measures utilisation of resources — how busy the resources are kept.

Parallel system performance of a system consisting of n processors is often characterised using speedup — the serial execution time divided by parallel execution time: $speedup(n) = time(1)/time(n)$. System efficiency is defined as the speedup divided by the number of processors:

$$\textit{system efficiency(n)} = speedup(n)/n$$

Thus efficiency is maximised at 1.0 when n=1.

From Figure 5.7, it can be seen that for the case of 32 GB, when more than 24 nodes are used, system efficiency goes down to below 50% of full efficiency and reaches a low minimum of 40% when 32 dynamic nodes are used. Therefore, at the 24 node point adding more nodes decreases indexing time but utilisation per machine decreases to levels where each machine does substantially less work, with, for example, each machine doing under 60 seconds of indexing time. When system efficiency falls to such low levels, it is of little advantage to use additional machines. For the 128 GB and 256 GB cases, system efficiency also declines with increasing numbers of dynamic nodes. However, due to the increased workload the system remains relatively efficient, reaching a minimum of 67% and 68% efficiency respectively.

This experiment has shown that for a given workload, the number of dynamic nodes can be increased to index the collection in the shortest possible time. However, adding more nodes to the grid in order to achieve the shortest indexing time can result is poor utilisation of resources with system efficiency falling to levels below 50%. Therefore, the optimal number of dynamic nodes is

the one that results in lowest indexing time at a system efficiency above a defined threshold.

During indexing, the dynamic indexers in conjunction with the storage servers store partial indices corresponding to each data chunk as a separate index. No index merging takes place. Lucene has the ability to search over multiple indices — however, the larger the number of indices, the larger the number of disk seeks that have to take place during querying. This can lead to long query response times. Therefore, the indices have to be merged into a single index before searching. The focus of the next experiment was on how index merging affects indexing time.

5.3.2 Index Merging

The prior accumulated indexing times reported do not include the time to merge indices. Index merging is a task of combining posting lists belonging to the same term into a single posting list.

Aim

The aim of this experiment was to determine how the merging step affects the accumulated indexing time.

Methodology and Results

The time to merge indices for various sub-collections of the AC.UK collection was measured. The AC.UK collection was used as is and not duplicated to simulate larger collection sizes. Duplicating the data collection only increases posting lists in size but does not change the size of the dictionary and this can affect merging time.

The results of the experiment are in Figure 5.8

Discussion

Figure 5.8 shows accumulated indexing time with and without merging time when using a single dynamic indexer. It is clear that the impact of overall indexing time is relatively small, with merging time contributing up to a maximum of 2.8% of indexing time. With multiple indexers and multiple storage servers, indexing time is reduced, but also the storage servers merge the indices in parallel and thus merging time is reduced also. Therefore, index merging time does not significantly increase the accumulated indexing time.

The focus of the experiments in this section was on dynamic indexers and their performance. Due to their unpredictable nature, dynamic nodes can only be used for indexing. Other tasks that require high availability such as storage of indices can only be performed by the more predictable

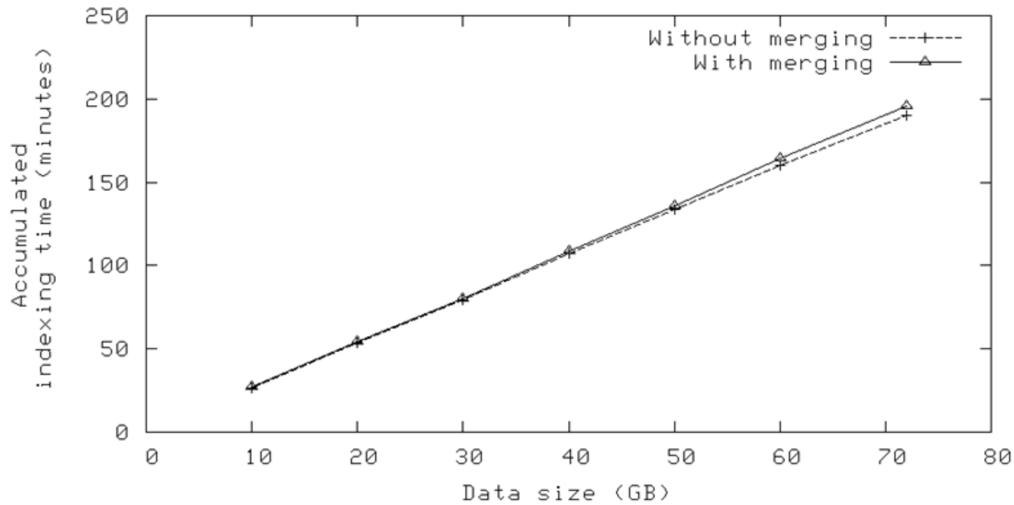


FIG. 5.8: Indexing time inclusive and non-inclusive of merging time

dedicated nodes (storage servers). So far the storage servers have only been used for the purpose of storing indices. However, from the CPU load average it is clear that the storage servers were under-utilised. The next experiment focused on storage servers and how they could be better utilised.

5.3.3 Utilizing Storage Servers

The CPU load average on the storage servers shows that in comparison to the dynamic indexers, the storage servers are relatively under-utilised. Thus the storage servers could be better utilised by also running indexing jobs on them. Furthermore, when the chosen approach is Non-local Data indexing, the storage servers will also need to store the source files that need to be indexed.

For each type of distributed indexing, the storage servers can be employed to do two different combinations of tasks.

1. Local Data indexing
   (a) **Indices**. The storage servers only serve the purpose of storing indices.
   (b) **Indexing + Indices**. The storage servers index data in addition to storing indices.
2. Non-local Data indexing
   (a) **Source files + Indices**. The storage servers only serve the purpose of storing source files and indices on SRB.
   (b) **Source files + Indexing + Indices**. The storage servers index data in addition to storing source files and indices.

Aim

The aim of this experiment was to determine, for each type of distributed indexing, the combination of tasks that leads to fastest indexing time while not overwhelming the resources.

Methodology and Results

Indexing time was measured for the different combinations of tasks. The accumulated indexing time, indexing time and communication and storage time were measured as before.
The results of the experiment are in Figure 5.9, Figure 5.10 and Figure 5.11

Discussion

Figure 5.9(a) shows that the indexing time behaviour for the cases of `Indices` and `Indexing + Indices` follows the same trend. As discussed in 5.3.1 this trend shows a decrease in indexing time as the number of dynamic indexers increases, with indexing time approaching a constant value.
The difference between the `Indices` and `Indexing + Indices` cases is that the latter results in faster indexing. This is due to the fact that the storage servers also perform indexing in addition to storing indices, thus effectively increasing the number of indexers.
As shown in Figure 5.9(a), for the case of `Source files + Indexing + Indices`, indexing time only decreases up to when 16 dynamic nodes are used — after that indexing time remains more or less the same. A breakdown analysis of the accumulated indexing time, Figure 5.9(b), helps explain this behaviour. It can be seen that after the case of 16 dynamic nodes, communication and storage time is higher than the time taken to index data. Any performance gain resulting from the use of increased dynamic indexers is canceled out by the increase in communication time.
The high communication time incurred in the case of `Source files + Indexing + Indices` is a result of the need to transfer the source files over a link that only attains SRB write and read rates of approximately 9 MB/s. As the number of dynamic indexers increases, communication time goes up because the number of requests to the SRB metadata catalog and to the SRB servers increases, resulting in performance degradation.
The CPU load shown in Figure 5.9(c) shows that the storage servers are under-utilised in the case of `Indices` whereas in the case of `Indexing + Indices`, the storage servers are well-utilised without overwhelming them, with the CPU load remaining close to the number of CPUs in the machine. The CPU load graph also shows that when the storage servers are performing all three tasks simultaneously, the machines are overwhelmed with the CPU load being nearly double the number of CPUs.

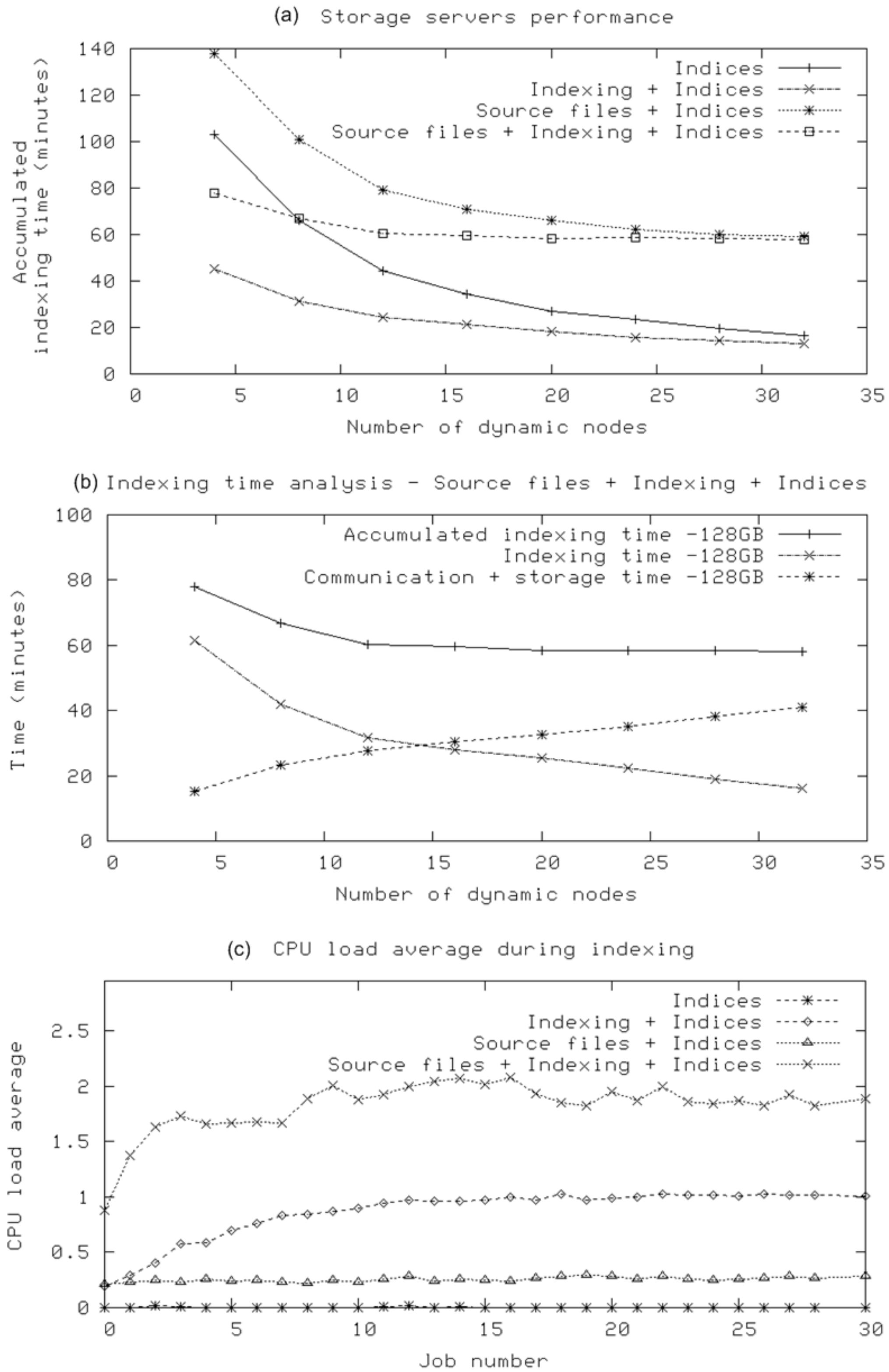


FIG. 5.9: Performance of storage servers. The data size used is 128 GB.

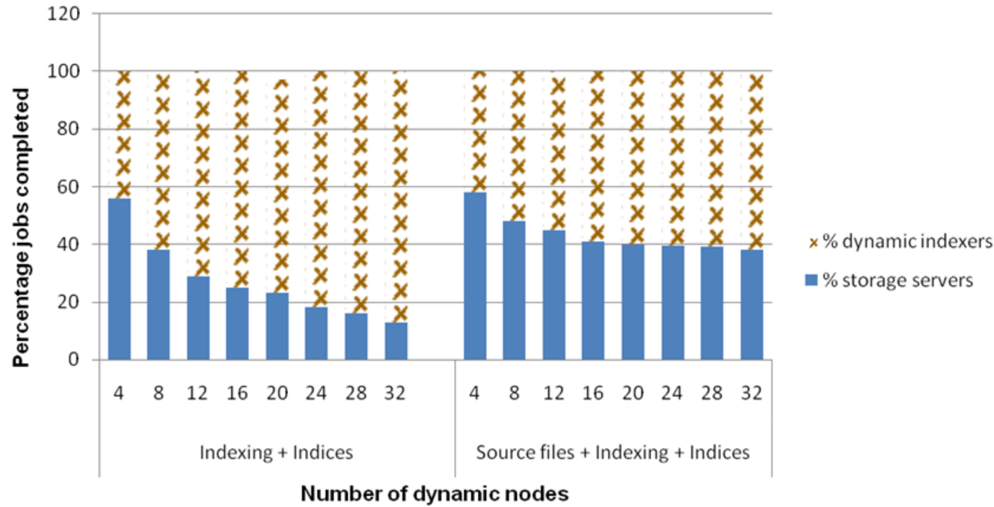


FIG. 5.10: Job completion comparison for the case `Indexing + Indices` and the case of `Source files + Indexing + Indices`. The Data size used is 128 GB, 6 storage servers were used in both cases.

Figure 5.10 is a job completion comparison between the `Indexing + Indices` and `Source files + Indexing + Indices`. This is further evidence that the latter case spends more time in communication. This results in dynamic indexers only completing a relatively small portion of the jobs in comparison to the `Indexing + Indices` case, despite the fact that dynamic indexers outnumber the storage servers. Figure 5.11 shows that increasing the number of storage servers can alleviate this problem.

This experiment has shown that for the Local Data indexing approach, it is beneficial to use the storage servers for indexing in addition to storing indices. However, for the Non-local Data indexing approach, using the storage servers to index data in addition to serving source files and indices overwhelms storage servers. Thus for the Non-local Data indexing approach it is better to leave the storage servers to serve the source files and indices.

Experiments carried out thus far have shown how the number of dynamic nodes can be chosen to achieve the required performance and how dedicated nodes can be best utilised. One aspect related to search engine indexing whose evaluation has not been discussed is the ability to perform index updates. The next experiment focused on index updating.

5.4 INDEX MAINTENANCE

Although not as highly dynamic as the Web, intranets are not static environments either. Data collections change over time, new documents are added, obsolete documents are deleted and

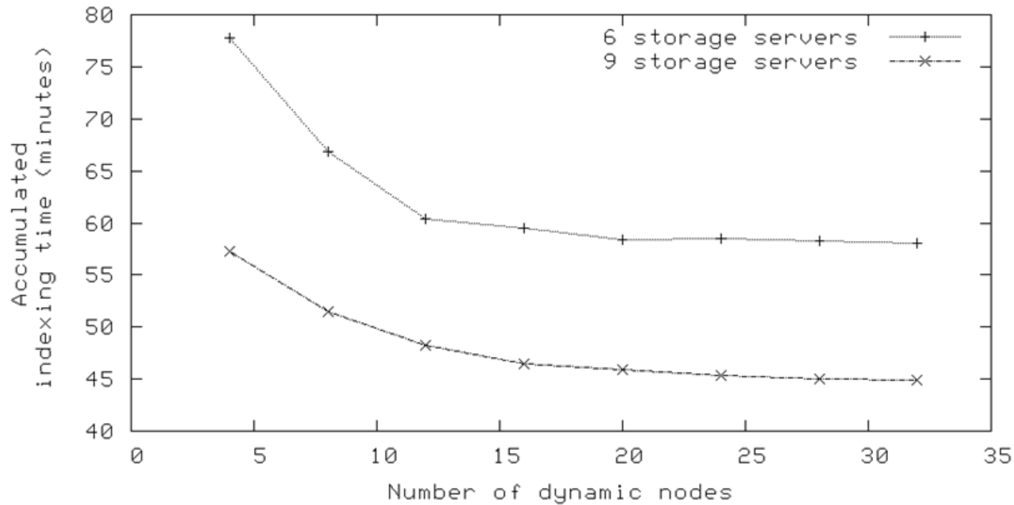


FIG. 5.11: Performance of storage servers. Data size used is 128 GB, 6 and 9 storage servers used.

documents are modified. Such changes in data collections need to be reflected in the search results in as short a time as possible. This experiment evaluated performance of updating the index.

Lucene employs a merge-based update strategy which combines the in-memory index with the on-disk index in a sequential manner.

Aim

The aim of this experiment was to determine if the index can be updated incrementally and, if so, how long it takes to perform index update operations — document additions, deletions and modifications. The experiment also aimed to determine if these index update operations can be optimised.

Methodology and Results

The metric of interest is the time taken to update an existing index. The experiments were carried out on the UCT.AC.ZA collections. The first part of the experiment evaluated performance for document additions only, allowing new documents to be added but not allowing existing documents to be deleted or modified. The average time to insert a document into the index was measured for varying index sizes.

The second part of the experiment focused on document deletions and modifications. The number of documents buffered in memory before writing them to disk was varied and total time to update the index was measured with respect to the document buffer size.

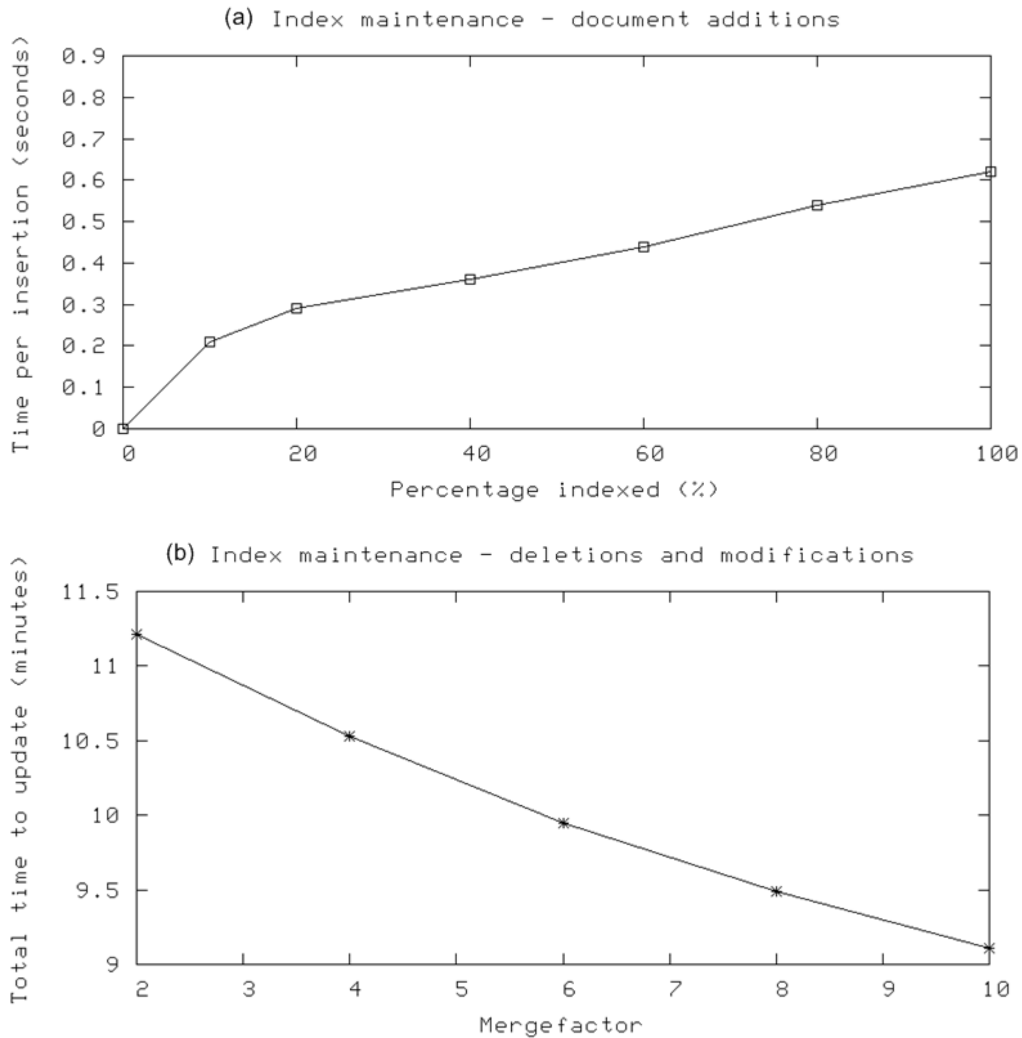


FIG. 5.12: Index maintenance performance. The mergefactor controls the number of documents buffered in memory.

Discussion

Figure 5.12(a) shows the average time taken per document insertion for varying percentages of the UCT.AC.ZA1 collection. The time to add a document is less than 0.1 seconds when 10% of the collection is indexed and increases up to about 0.6 seconds when 100% of the collection is indexed. The time per insertion increases in a near logarithmic manner as the data collection increases. This performance is similar to the index update performance reported in [12], which shows a logarithmic increase in document insertion time with increase in data size.

Figure 5.12(b) shows the total time to perform document deletions and modifications on an existing index. Unlike document additions, the operations of deletions and modifications have to be performed on the entire collection because they require the existing index to have the entries of

the obsolete versions of the documents to be deleted or modified. These obsolete versions are then deleted or replaced, as required. It can be seen that the time taken to perform document modifications and deletions of the UCT.AC.ZA1 collection as reflected in UCT.AC.ZA2 is about 11.2 minutes. However, by increasing the mergefactor, the total update time is reduced by up to 19% when the mergefactor of 10 is used. The mergefactor determines how many document updates are buffered in memory before writing the changes to the original index.

This experiment has shown that the index can be incrementally updated on hybrid scavenger grid. Documents insertion time increases in a near logarithmic manner as data size increases. Furthermore, one way of optimising index update is by varying the mergefactor.

Experiments carried out thus far have shown indexing performance of the hybrid scavenger grid. The question to ask at this stage is how performance of the hybrid scavenger grid compares to other architectures and whether it is worth investing in dedicated nodes and maintaining a grid, if the cost is the same as that of a middle or high end multi-processor server which has comparable performance.

5.5 HYBRID SCAVENGER GRID VERSUS MULTI-CORE

Processor designers are seeing the end of Moore's Law [84]. Moore said that the number of transistors on a chip doubles approximately every 18 months. The effect of Moore's law is that each new chip design could be expected to do twice as much as its predecessor leading to an exponential rise in performance matched by a corresponding fall in the cost of computing power.

Despite progress in the ability to fit more transistors on a die, clock speeds have seen slow progress in recent years. Power management is a big problem — modern chips contain a lot of transistors such that moving the resulting heat off the processors is increasingly difficult. Consequently, as the number of transistors on a chip increases, the clock speed remains flat or declines. For this reason, chip architects have turned to multi-core architectures whereby multiple processor elements are placed on one chip.

At the time of writing, two core processors are already common in desktop computers, with four and eight cores found in an increasing number of middle to high-end servers. While cost-effective scalability is one of the advantages of a hybrid scavenger grid-based search engine, the challenge is the process of designing, implementing and running a search engine on such a distributed system. Limitations of the grid such as the unpredictable nature of dynamic nodes and job failure rate can hinder performance. Thus it can be argued that with the advent of multi-core technology, powerful servers can be purchased for low prices and thus the centralised architecture should be the architecture for workloads of certain magnitudes.

Aim

This experiment compares performance of the hybrid scavenger grid to that of a multi-core architecture. It seeks to find the circumstances under which each system is the more beneficial choice with regards to performance, cost-effectiveness and system efficiency.

Methodology and Results

Indexing performance was measured on the quad-core machine by running jobs on each of its 4 cores simultaneously. The cost-effectiveness and system efficiency of the quad-core machine was then compared to that of the hybrid scavenger grid. Given two systems both doing the same task, the cost-effective system is the system that maximises job throughput per unit cost or, equivalently, the system that minimises the cost/performance (cost divided by performance) value of the system [96]. System efficiency measures utilisation of resources. A highly utilised system has high system efficiency.

The two systems can be defined as follows:

$$Grid(s,d) \text{ and } Multicore(c)$$

Where `s` and `d` in `Grid(s,d)` are the number of storage servers and the number of dynamic indexers respectively. The `c` in `Multicore(c)` is the number of cores the machine has.

As discussed in before, parallel system performance of a system consisting is often characterised using speedup. System efficiency is then defined in terms of speedup (see 5.3.1). Let the serial execution time on a single machine be `time(1)`, the time to execute on multi-core server be `time(Multicore(c))` and the time to execute on the grid be `time(Grid(s,d))`.

For the grid and multi-core, system efficiency is:

$$\textit{system efficiency (Grid(s,d))} = speedup(Grid(s,d))/(s+d) \tag{5.2}$$

$$\textit{system efficiency (Multicore(c))} = speedup(Multicore(c))/(c) \tag{5.3}$$

Grid system efficiency (Equation 5.2) assumes that the storage servers also are used as indexers. If only dynamic nodes are used as indexers then speedup is divided by the number of dynamic nodes only.

Let the cost of a single machine be `cost(1)`, the cost of the grid be `cost(Grid(s,d))` and, the cost of the multi-core machine be `cost(Multicore(c))`. To determine the cost-effectiveness of a system with n processors which cost $cost(n)$, performance and cost are combined to obtain cost/performance [96]:

$$costperf(n) = \frac{cost(n)}{1/time(n)}$$

$$costperf(Grid) = \frac{cost(Grid)}{1/time(Grid(s,d))} \tag{5.4}$$

$$costperf(Multicore) = \frac{cost(Multicore)}{1/time(Multicore(c))} \tag{5.5}$$

A system is more cost-effective than the other when its costperf value is smaller than the other system's. The cost of a system depends on one's point of view. It can be hardware cost for processors, memory, I/O, power supplies, and so forth, and it can also include software costs of middleware and other system software involved. For the purpose of this experiment, the cost only includes processor cost. The prices are list prices in US dollars (as of 7 December, 2008)[47]. The processor (Intel Quad-Core Xeon X5472/3 GHz) in the quad-core machine costs $1,022 and a typical desktop processor (Intel Core 2 Duo E8400/3 GHz)[1] costs $163.

Using these prices, the costs of the multi-core and grid are:

$$cost(Multicore(4)) = 1,022$$

$$cost(Grid(s,d)) = 163 * s$$

To get the cost of the grid, $cost(Grid(s,d))$, the cost of one storage server is multiplied by the total number of storage servers, s. In these experiments 6 storage servers were used. Using the ratios of the prices [96], the resulting cost functions are:

$$cost(Multicore(4)) = 6.27 \tag{5.6}$$

$$cost(Grid(s,d)) = s \tag{5.7}$$

The cost of the grid only involves the cost of the dedicated nodes because the dynamic nodes are obtained from cycle-stealing and thus they could be considered as having a minimal or external cost.

The results of the experiment are shown in Figure 5.13, Figure 5.14 and Figure 5.15. The system efficiency values shown in Figure 5.13(b), Figure 5.14(b) and Figure 5.15(b) were computed using Equation 5.2 and Equation 5.3. The cost/performance values shown in Figure 5.13(c), Figure 5.14(c) and Figure 5.15(c) were computed using Equation 5.4 and Equation 5.5.

[1]The experiments used Pentium 4 machines, however these are no longer listed in the price list from Intel, current new desktop computers typically have an Intel Core 2 Duo processor and thus the price of a Core 2 Duo was used.

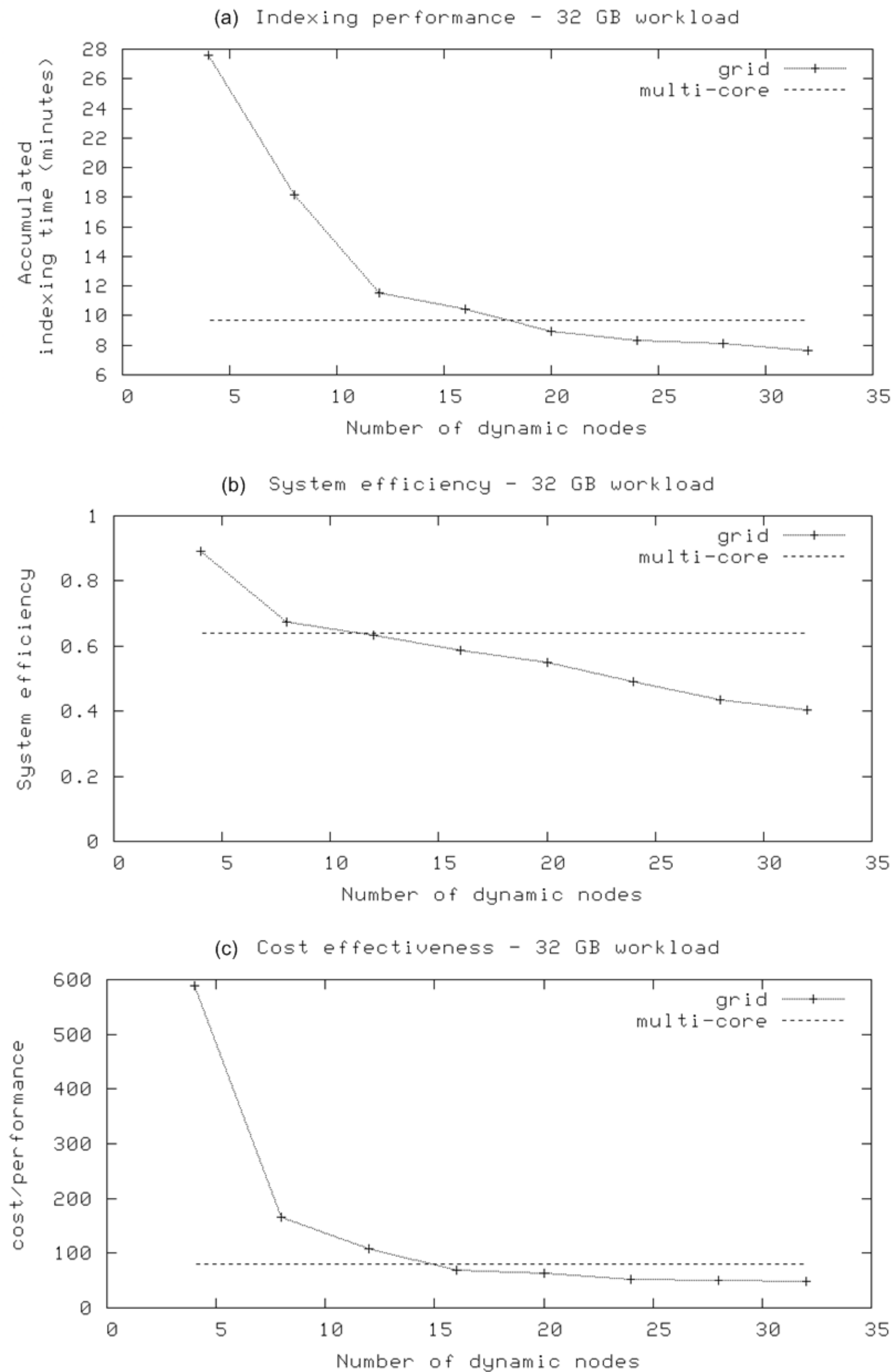


FIG. 5.13: Performance, system efficiency and cost-effectiveness: 32 GB

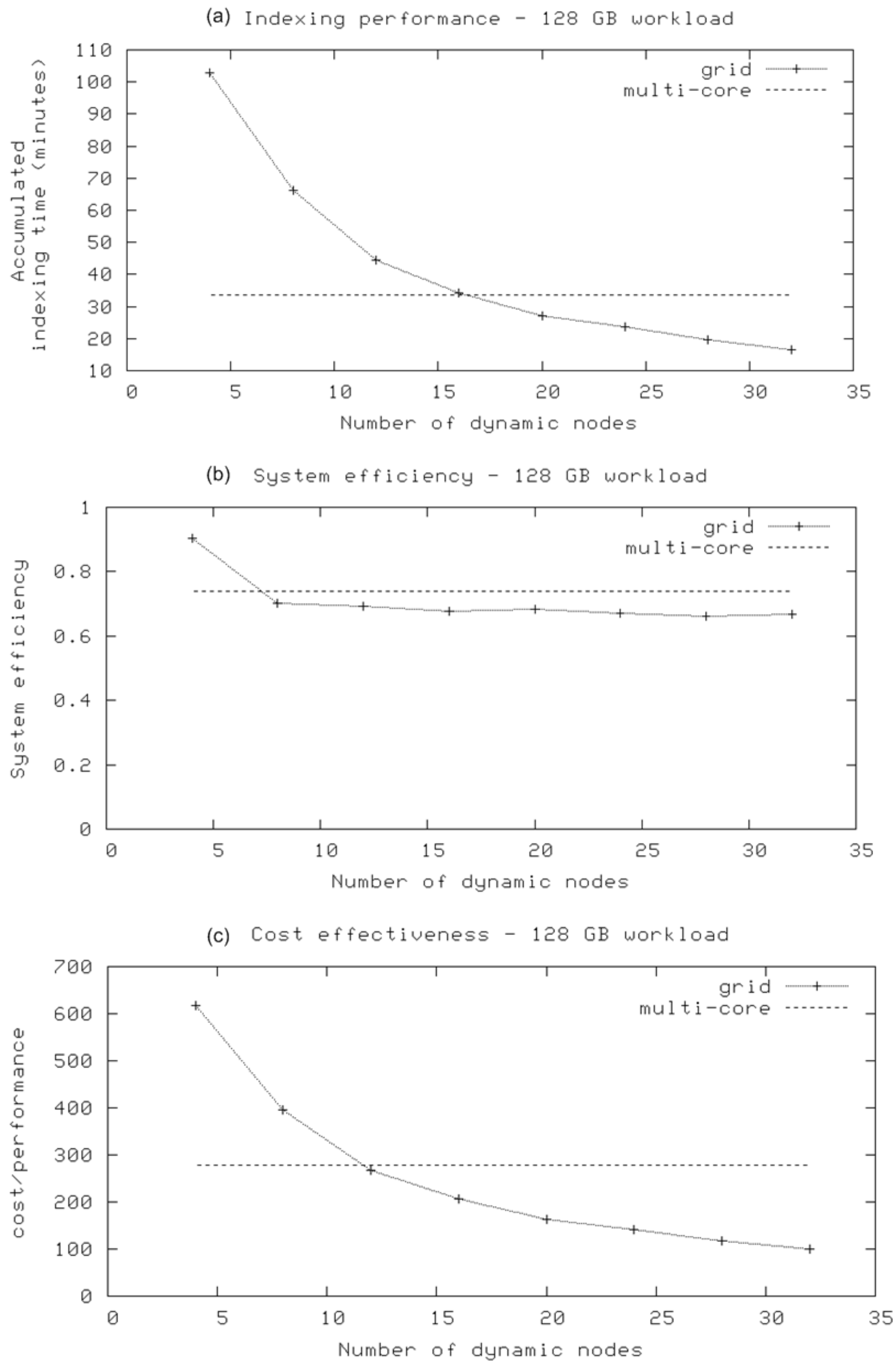


FIG. 5.14: Performance, system efficiency and cost-effectiveness: 128 GB

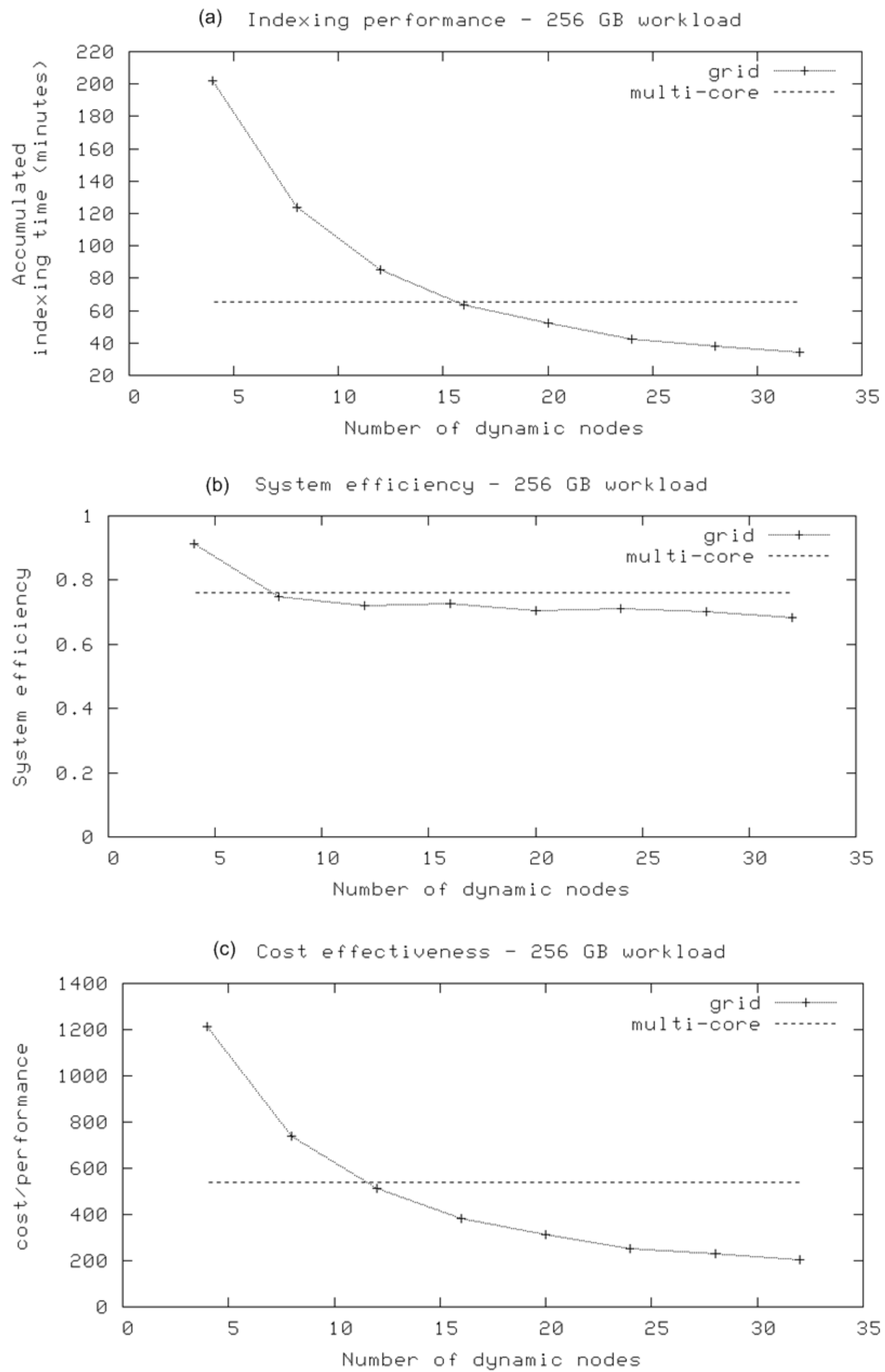


FIG. 5.15: Performance, system efficiency and cost-effectiveness: 256 GB

Discussion

Figure 5.13 shows the indexing, performance, system efficiency and cost-effectiveness of both systems, for the workload of 32 GB. The system efficiency of the multi-core is constant at 0.64, whereas that of the grid varies with the number of dynamic nodes. It can be seen that for more than 12 dynamic nodes, the efficiency of the grid is lower than that of the multi-core, and continues to decline as more nodes are added. The performance graph (Figure 5.13(a)) shows that the grid performs better than the multi-core machine when 16 or more dynamic nodes are used. It can also be seen that the performance (Figure 5.13(a)) and cost-effectiveness (Figure 5.13(c)) of the grid are only significantly better than those of the multi-core when 24 or more nodes are used. However, at this point the efficiency(Figure 5.13(b)) of the grid is 0.49 whereas that of the multi-core is 0.64. Therefore, for this particular workload it can be concluded that multi-core is a better choice since employing the grid leads to poorly utilised resources.

Figure 5.14 shows the indexing performance, system efficiency and cost-effectiveness of both systems for the workload of 128 GB. The system efficiency of the multi-core is 0.74. For more than 8 dynamic nodes, the efficiency of the grid is lower than that of the multi-core but it remains relatively high even as the number of dynamic nodes increases, degrading to a minimum of 0.67 for 32 dynamic nodes. Figure 5.14(a) shows that the grid performs better when 16 or more dynamic nodes are used. At that point the grid has a system efficiency of 0.69 which is 5% less than that of the multi-core. Also at that point the grid is more cost-effective than the grid. At 32 dynamic nodes, the cost-effectiveness of the grid is more than twice that of the multi-core and the grid system efficiency is at 0.67, which is 7% less than that of the multi-core. Thus for this workload, it can be concluded that the grid is more cost-effective and at the same time utilisation of the grid resources is relatively high.

The results of the 256GB workload are shown in Figure 5.15. The system efficiency of the multi-core is 0.76. The efficiency of the grid is lower than that of the multi-core when more than 8 dynamic nodes are used — it remains relatively high and reaches a minimum of 0.68 for 32 dynamic nodes. It can be seen in Figure 5.15(b), that the grid performs better and is more cost-effective when 16 or more dynamic nodes are used. At that point the grid has a system efficiency of 0.73 which is 3% less than that of the multi-core. For this workload, it can be concluded that the grid is more cost-effective and at the same time utilisation of the grid resources is relatively high.

In these experiments the focus was on varying dynamic nodes and not the static nodes mainly because in many cases the number of static nodes are few and cannot be controlled. Typically, dynamic nodes are available in large numbers and it is important to determine when to stop adding more to the system. It is apparent that the static nodes can also be varied to establish the

number of static nodes that achieves the best performance.

This experiment has shown that for small workloads, although the grid provides better performance and cost-effectiveness for large numbers of dynamic nodes, the system efficiency goes to low levels that render the usefulness of the grid questionable. For modest to large workloads, the grid is a more beneficial approach achieving better cost-effectiveness and maintaining relatively high system utilisation.

Having established that for modest to large scale workloads the hybrid scavenger grid is a beneficial architecture for search engine indexing, it important to also evaluate its performance for searching.

5.6 QUERYING PERFORMANCE ANALYSIS

The ultimate goal of a search engine is to respond to queries fast. Search results should be returned within at most a few seconds, preferably less, similar to what users of Web search engines are accustomed to. The underlying index structure that is searched during querying dictates query response times. An efficient structure does not require a lot of disk access especially not disk seeks. Frequent and inefficient disk access patterns can substantially degrade query response time performance.

In a scalable search engine, query response time should remain more or less constant even as the size of the searched index increases. Moreover, the index distribution among the index storage servers should enable query response times to be more or less the same for different queries — the time to respond to individual queries should not be substantially longer for some queries while it is shorter for others.

Aim

This experiment measured query throughput with regards to the size of the index. The aim was to determine if query response time remains constant regardless of index size. The experiment also aimed to find if queries can be responded to within sub-second time and if query response times do not substantially differ for different queries.

Methodology and Results

For the first part, a client program submits all the queries to the scheduler simultaneously. Each query is submitted without waiting for the results. At the query servers, queries are handled by separate threads in parallel. The total time taken to respond to all the queries is recorded and the

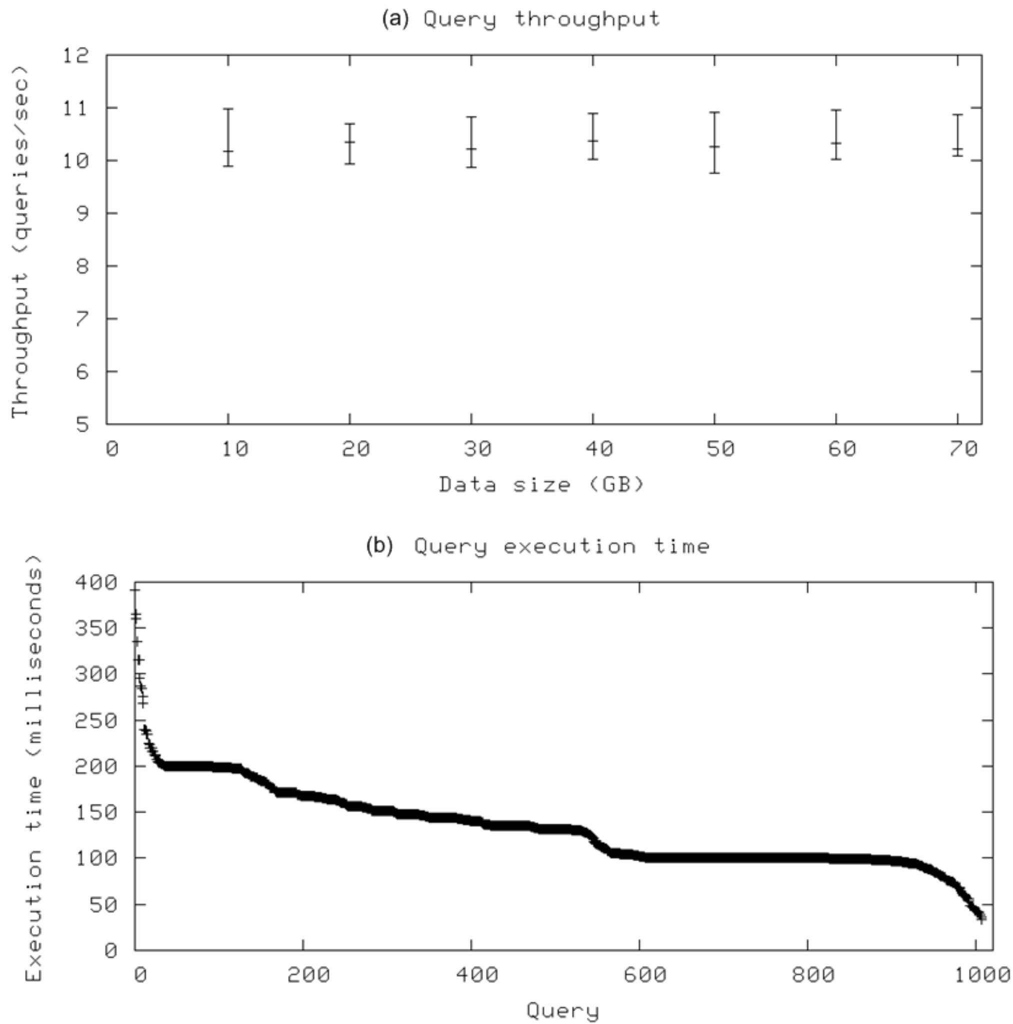


FIG. 5.16: Querying performance with 6 storage/query servers.

throughput is computed.

For the second part, the execution time to respond to each query was measured by sending one query at a time. The client submits the query to the scheduler and waits for the results before sending the next query. This is done for all the queries in the query set. Execution of queries in isolation is done in order to ensure that the recorded time is independent of factors such as interference among the various processing threads at the query servers. The results of the experiment are in Figure 5.16.

Discussion

From the Figure 5.16(a) it can be seen that the average query response time remains fairly constant even as the data size is increased. Query throughput is determined by the performance of the

query servers and also by the arrival rate of queries at the scheduler [7] . The attained average throughput is 10.27 queries per second. This means that the 6 storage servers used can process up to 36,972 queries per hour or roughly close to a million queries per day. With the average throughput of 10.27, the average time per query is 0.10 seconds. The query throughput attained is comparable to that of other distributed systems. Badue et al [7] reported that with a cluster of 8 machines, they observed a query throughput of 12 queries per second, with an average time per query of 0.12 seconds.

Figure 5.16(b) shows that response times for all the queries is below one second, with an average query response time of 0.13 seconds, a minimum of 0.034 seconds and maximum of 0.39 seconds. This experiment has shown that the average query response time remains fairly constant, that query response times are below one second and that the variance in query response times is not substantial. The storage servers download the data to their local disk before responding to queries. SRB file streaming methods have proven to be slow and resulted in slow query response times. In their implementation of a grid-based version of Lucene, Meij and de Rijke [60] made the same observation about the file streaming capabilities of SRB. Thus SRB only served as a way to store the indices that allows both dynamic nodes and dedicated nodes to store and access the distributed indices seamlessly.

5.7 SUMMARY

The results of the experiments can be summarised as follows.

1. Grid middleware overhead. SRB read/write speeds largely depend on network speed. In the fast network with 75 MB/s bandwidth, the overhead in indexing time resulting from the presence of SRB is relatively low. In a slow network of 12 MB/s bandwidth, SRB incurs a high overhead on indexing time which results in long indexing times when using the Non-local Data indexing approach.

2. Varying grid composition. The optimal number of dynamic nodes to index a collection of a given size depends on the desired level of system efficiency. Increasing the number of dynamic nodes reduces indexing time but may also lead to degraded system efficiency. The optimal combination of tasks performed by the storage servers depends on the distributed indexing approach used. For the Local Data indexing approach, it is beneficial to use the storage servers for indexing and storing indices. However, for the Non-Local Data indexing approach, allocating the three tasks of indexing, source file storage and index storage to the storage servers overwhelms the storage servers to a point where the CPU load average becomes twice as much as the number of processors.

3. Index maintenance. The index can be incrementally updated. New documents can be added to an existing index with a near logarithmic performance degradation. Furthermore, performance of updates, including document additions, modifications and deletions can be improved by using bigger buffer sizes. The buffer size controls how many document updates are stored in memory before merging them with the on-disk index.

4. Hybrid scavenger grid vs multi-core. For small workloads, the grid architecture can deliver better throughput per unit cost in comparison to the multi-core architecture, but it results in poor utilisation of resources. Therefore, since performance of the multi-core for small workloads is not much worse than grid performance, the multi-core can be used as the architecture of choice for small workloads. For workloads of modest to large sizes, the grid architecture delivers better throughput per unit cost than multi-core by wider margins. At the same time, the grid system efficiency is not much lower than that of the multi-core system.

5. Querying performance analysis. Querying time does not vary significantly even as the data size increases. The average query response times for the queries used was 0.13 seconds. Therefore, the architecture can realise sub-second query responses for modest data sizes.

Therefore, from the results of the evaluation, it can be argued that the hybrid scavenger grid architecture is cost-effective and efficient for modest to large workloads and can realise sub-second query responses.

The experiments listed here are only a small sample of experiments that can be carried out to assess the effectiveness of the hybrid scavenger grid approach as an underlying architecture for search engine operations. Other possible tests are suggested in the future work section of chapter 6.

CHAPTER

6

Conclusion

Rapid increase in data collections means that high quality search services are not only needed on the Web but also within restricted environments such as organisational intranets. Large scale informational retrieval operations have high computational needs and thus require scalable hardware architectures. Cluster computing is one such architecture. The main downside is the high cost of acquiring the large number of dedicated resources needed to create a scalable cluster and also the cost of operating clusters. The hybrid scavenger grid proposed in this thesis only requires a small number of dedicated resources that are augmented with idle desktop workstations. Dedicated resources are required to provide reliability and predictability to the architecture whereas the dynamic nodes make the architecture scalable.

The hybrid scavenger grid proves to be a feasible architecture for a search engine that supports medium- to large-scale data collections. The architecture reduces indexing time and responds to queries within sub-seconds. The resources of the architecture can be organised in way that delivers the best performance by using the right number of nodes and the right combination of tasks that the nodes perform. The desired levels of performance and system efficiency determine the optimal number of dynamic nodes to index a collection. Increasing the number of dynamic nodes reduces indexing time but also leads to degraded system efficiency. The combination of tasks that delivers the best performance depends on the type of distributed indexing used. For Local Data indexing, the combination of using both dynamic nodes and dedicated nodes for indexing and using the dedicated nodes for storing indices delivers the best performance. For the Non-local Data indexing approach, the combination of using only dynamic nodes for indexing and using the dedicated nodes for storing source data files and indices delivers the best performance.

For small workloads, although the grid architecture delivers better throughput per unit cost in comparison to a quad-core machine, it results in poor utilisation of resources. For workloads

of modest to large sizes, the grid architecture delivers better throughput per unit cost than the quad-core by wider margins at a system efficiency that is comparable to that of the quad-core.

The implementation of a distributed search engine, the deployment of the search engine on a hybrid scavenger grid and its evaluation have collectively shown that:

1. It is beneficial to use a hybrid scavenger grid architecture for indexing and searching as it
   (a) reduces indexing time
   (b) realises sub-second query response times
2. The right resource composition of the grid can be achieved to get the best performance by using the right number of nodes and scheduling the right combination of tasks on the nodes.
3. Compared to a multi-core machine of 4 cores, the hybrid scavenger grid delivers better performance and is more cost-effective and efficient for modest to large scale workloads.
4. The hybrid scavenger grid architecture can scale to large collections of data by adding more dynamic nodes as required.

Small companies, academic institutions and other organisations can reduce costs by using a hybrid scavenger grid architecture as the underlying hardware infrastructure for their search services. Moreover, the hybrid scavenger grid advocates the use of resources that are already present within the organisation, thus it could empower organisations in the developing world to provide their own search services with limited but well-utilised computing resources. Developing world organisations can invest in a small number of dedicated resources and the rest of the computational power can be obtained from the dynamic resources. In the developing world, however, some institutions avoid leaving their computers switched on overnight in order to reduce power consumption. In such a case, certain periods of time (for example, certain weekdays or weekends) can be chosen during which the idle desktop machines can be left switched on.

Limitations

From performance evaluation, it is apparent that the Non-local Data indexing approach performs poorly. With Non-local Data indexing, the nodes download source data that is stored on the storage servers on SRB. The extra step of interacting with SRB incurs network time that adds to the accumulated indexing time. Within a fast network, the overhead is relatively low (8-10% increase in indexing time within a network of maximum available bandwidth of 75 MB/sec). However within a slow network the overhead is high (39-41% increase in indexing time within a network of maximum available bandwidth of 12 MB/sec). Due to this high overhead in slow networks,

downloading the files from SRB degrades indexing performance significantly. The alternative method of using SRB file streaming methods to directly access and index data stored on the SRB filesystem delivered worse performance as the SRB file streaming methods were slow and error prone.

While this work was underway, a new generation of data management grid middleware, called i Rule Oriented Data Systems (iRODS), was developed by the developers of SRB at the San Diego Supercomputer Center. SRB has been said to be a great concept that is poorly engineered [94], thus iRODS might have addressed some of SRB's implementation shortcomings. The file streaming methods of iRODS could be evaluated in the context of this project. Another option is also to explore and evaluate performance of other data grid middleware solutions such as the GridFTP data management component of the Globus Toolkit which has been found[94] to have a robust means of file transfer in comparison to SRB. The Globus Toolkit was not chosen for this project because of the many other services that come with the toolkit that were not needed for this project. The Globus Toolkit was meant to be modular but it is not clear how to separate one service from the rest.

With the implementation presented in this thesis, if the chosen approach is Non-local Data indexing and if the network speed is slower than 12MB/sec then the overhead incurred by SRB will be even higher, resulting in overall poor performance of the grid. The network becomes a bottleneck to the centrally stored data. Therefore it would be beneficial for limited bandwidth environments to use the fully distributed approach — Local Data indexing — even when the data is stored on a few centralised servers. The data can be staged on dynamic nodes before indexing begins by distributed the data among the nodes that are involved in indexing. Another option that naturally results in distributed data is the use of a parallel crawler which runs on separate machines simultaneously. The resulting distributed data can then be indexed using the Local-Data approach — each machine indexes the data downloaded by its crawler.

Ideally, experiments should be conducted on a standard collection to allow accurate repeatability of experiments. The dataset used in the experiment is not a standard data collection mainly because the author had no access to standard collections such as the GOV2 TREC collection which require acquisition costs.

Future Work

There are several open questions that have not been answered by this thesis, in particular those related to distributed search engine algorithms. The result merging algorithm used re-ranks the partial result lists retrieved by each server in order to make the document scores from the different servers comparable before merging them into a single list. The document scores are not immedi-

ately comparable because each server computes similarity scores between documents and queries using local collection statistics. The documents are assigned new scores which are their original scores altered by a collection weight for their respective collections. The collection weights used are based on collection scores that take into account the length of the result list retrieved by the server holding the collection. The re-ranking approach is an alternative to using global collection statistics of the collection as a whole. The global collection statistics are computed periodically to reflect changes in the collection and each server is updated with the new values. An analysis of the quality of ranking of the two methods can be explored. Testing with a standard collection can enable the precision graphs of the two methods to be compared.

The dedicated machines that store the indices and respond to queries can fail independently. The indices can therefore be duplicated among the storage servers. In the case where a server fails, the servers holding the duplicate index of the index that the failed server was serving queries with can begin serving queries using both its own index and the index of the failed server. Lucene has the ability to search across multiple indices, therefore this would be a straightforward fault-tolerance extension.

The experiments conducted are a subset of possible experiments that could be carried out to evaluate the effectiveness of the hybrid scavenger grid. Other possible experiments include:

Ease of Maintenance. Maintaining a distributed system requires significant human effort. Large data collections of several terabytes of data require a large grid consisting of large numbers of dynamic nodes. As the size of the grid grows, the effort required to operate and maintain the grid also becomes greater. Therefore, it would also be of interest to know the human cost of a hybrid scavenger grid operation in comparison to the other architectures while taking into account performance, hardware cost-effectiveness and resource efficiency.

Node failure rate. The search engine developed handled node failure by reallocating the work of unresponsive nodes. Experiments showing how performance is affected by varying node failure rates could also be carried out.

Network Utilisation. The system efficiency analysis took into account the utilisation of the machines of the grid but not that of the network connecting the machines. It would be important to also measure network utilisation on different segments of the network during indexing. Such experiments would reveal performance of the grid for different network traffic scenarios.

Bibliography

[1] Architecture of a grid-enabled web search engine. *Journal of Information Processing and Management*, 43(3):609–623, 2007. Available http://dx.doi.org/10.1016/j.ipm.2006.10.011.

[2] J. Allan. Information retrieval course. Website, 2007. University of Massachusetts Amherst,Department of Computer Science. http://ciir.cs.umass.edu/cmpsci646/.

[3] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster. The globus striped gridFTP framework and server. In *SC'2005 Conference CD*, page 205, Seattle, Washington, USA, 2005. IEEE/ACM SIGARCH. Available http://dx.doi.org/10.1109/SC.2005.72.

[4] A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan. Searching the web. *ACMTIT: ACM Transactions on Internet Technology*, 1, 2001. Available http://doi.acm.org/10.1145/383034.383035.

[5] S. Asaduzzaman. *Managing Opportunistic and Dedicated Resources in a Bi-modal Service Deployment Architecture*. PhD thesis, McGill University, October 2007. Available http://digitool.library.mcgill.ca:8881/thesisfile18713.pdf.

[6] Autonomy. Autonomy search engine. Website, 2008. http://www.autonomy.com.

[7] C. Badue, P. Golgher, R. Barbosa, B. RibeiroNeto, and N. Ziviani. Distributed processing of conjunctive queries. In *Heterogeneous and Distributed Information Retrieval workshop at the 28th ACM SIGIR Conference on Research and Development on Information Retrieval*, Salvador, Brazil, August 2005. Available http://hdir2005.isti.cnr.it/camera-ready/5.Badue.pdf.

[8] R. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri. Challenges on distributed web retrieval. In *ICDE*, pages 6–20, Istanbul, Turkey, 2007. IEEE. Available http://research.yahoo.com/files/baeza-yates2007icde-invited.pdf.

[9] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, March/April 2003.

[10] C. K. Baru, R. W. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In S. A. MacKay and J. H. Johnson, editors, *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative Research*, page 5, Toronto, Ontario, Canada, November/December 1998.

[11] J. Basney and M. Livny. Deploying a high throughput computing cluster. In R. Buyya, editor, *High Performance Cluster Computing*, pages 116–134. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[12] S. Büttcher. *Multi-User File System Search*. PhD thesis, University of Waterloo, 2007. Available http://uwspace.uwaterloo.ca/handle/10012/3149.

[13] R. Buyya. *Economic-based Distributed Resource Management and Scheduling for Grid Com-*

puting. PhD thesis, University of Monash, 2002. Available http://www.gridbus.org/~raj/thesis/.

[14] F. Cacheda, V. Plachouras, and I. Ounis. A case study of distributed information retrieval architectures to index one terabyte of text. *Information Processing and Management*, 41(5):1141–1161, 2005. Available http://ir.dcs.gla.ac.uk/terrier/publications/cacheda04case-study-terabyte.pdf.

[15] P. Callan, Z. Lu, and W. Croft. Searching distributed collections with inference networks. In *Proceedings of the Eighteenth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 21–28, Seattle, Washington, USA, July 1995. Available http://doi.acm.org/10.1145/215206.215328.

[16] B. B. Cambazoglu, A. Catal, and C. Aykanat. Effect of inverted index partitioning schemes on performance of query processing in parallel text retrieval systems. In A. Levi, E. Savas, H. Yenigün, S. Balcisoy, and Y. Saygin, editors, *Proceedings of the 21th International Symposium on Computer and Information Sciences - ISCIS*, volume 4263 of *Lecture Notes in Computer Science*, pages 717–725, Istanbul, Turkey, 2006. Available http://www.cs.bilkent.edu.tr/~aykanat/papers/06LNCS-iscis.pdf.

[17] D. Childs, H. Change, and A. Grayson. President-elect urges electronic medical record in 5 years. Website. http://abcnews.go.com/Health/President44/Story?id=6606536&page=1.

[18] J. Cho and H. Garcia-Molina. The evolution of the Web and implications for an incremental crawler. In *In Proceedings of the 26th International Conference on Very Large Databases*, Viena, Austria, 2000. Available http://oak.cs.ucla.edu/~cho/papers/cho-evol.pdf.

[19] Computerworld Inc. Storage power costs to approach $2B this year. Website, 2009. http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9102498&intsrc=hm_list.

[20] Condor. Condor high throughput computing. Website, 2007. http://www.cs.wisc.edu/condor/.

[21] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *The 4th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82. Springer-Verlag LNCS 1459, 1997. Available http://www.chinagrid.net/grid/paperppt/GlobusPaper/gram97.pdf.

[22] S. Das, S. Tewari, and L. Kleinrock. The case for servers in a peer-to-peer world. In *Proceedings of IEEE International Conference on Communications*, Istanbul, Turkey, June 2006. Available http://www.lk.cs.ucla.edu/PS/icc06servers.pdf.

[23] DILIGENT. A digital library infrastructure on grid enabled technology. Website, 2007. http://www.diligentproject.org.

[24] EPrints. Open access and institutional repositories with EPrints . Website, 2009. http://www.eprints.org/.

[25] F. E. Etim and U. Akwa-Ibom. Resource sharing in the digital age: Prospects and problems in african universities. *Library Philosophy and Practice*, 9(1), Fall 2006.

[26] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Information Systems*, 2(4):267–288, 1984. Available http://doi.acm.org/10.1145/2275.357411.

[27] FAST. FAST enterprise search. Website, 2008. http://www.fastsearch.com.

[28] FightAIDS@Home. Fight AIDS at Home. Website, 2007. http://fightaidsathome.scripps.edu/.

[29] S. Fitzgerald, I. T. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A directory service for configuring high-performance distributed computations. In *Proceedings 6th International Symposium on High Performance Distributed Computing (HPDC '97)*, pages 365–376, Portland, OR, USA, August 1997. IEEE Computer Society. Available ftp://ftp.globus.org/pub/globus/papers/hpdc97-mds.pdf.

[30] I. Foster. Globus toolkit version 4: Software for service-oriented systems. *Journal of*

Computer Science and Technology, 21(4), 2006. Available http://www.globus.org/alliance/publications/papers/IFIP-2005.pdf.

[31] I. Foster and A. Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2, 2003. Available http://iptps03.cs.berkeley.edu/final-papers/death_taxes.pdf.

[32] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1997. Available ftp://ftp.globus.org/pub/globus/papers/globus.pdf.

[33] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. 2002. Technical report, Global Grid Forum.

[34] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid, CCGRID*, pages 6–7. IEEE Computer Society, 2001. Available http://www.globus.org/alliance/publications/papers/anatomy.pdf.

[35] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. V. Reich. The open grid services architecture, version 1.0. Technical report, Global Grid Forum, 2005.

[36] V. I. Frants, J. Shapiro, I. Taksa, and V. G. Voiskunskii. Boolean search: Current state and perspectives. *Journal of the American Society of Information Science*, 50(1):86–95, 1999.

[37] S. Gerard. *The SMART Retrieval System - Experiments in Automatic Document Processing*. Prentice-Hall, Englewood Cliffs, NJ,USA, 1971.

[38] Globus. The Globus toolkit. Website, 2007. http://www.globus.org/toolkit.

[39] Google. The Google insights for search. Website. http://www.google.com/insights/search/.

[40] Google. The Google search appliance. Website, 2008. http://www.google.com/enterprise/index.html.

[41] D. Gore. Farewell to Alexandria: The theory of the no-growth, high-performance library. In D. Gore, editor, *Farewell to*.

[42] Grid IR. The grid IR working group. Website, 2007. http://www.gridir.org.

[43] D. Harman, R. Baeza-Yates, E. Fox, and W. Lee. Inverted files. In *Information Retrieval: Data Structures and Algorithms,*, pages 28–43. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1992.

[44] Harvard.edu. Africa higher education: student survey project . Website, 2009. http://www.arp.harvard.edu/AfricaHigherEducation.

[45] G. Haya, F. Scholze, and J.Vigen. Developing a grid-based search and categorization tool. *HEP Libraries Webzine*, January 2003. Available eprints.rclis.org/archive/00000446/02/developing_a_grid-based_search.pdf.

[46] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *Journal of the American Society of Information Science*, 54(8):713–729, 2003. Available http://dx.doi.org/10.1002/asi.10268.

[47] Intel Cooporation. Intel processor pricing. Website, 2009. http://www.intc.com/priceList.cfm.

[48] K. Spärck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21, 1972.

[49] C. Kenyon and G. Cheliotis. Creating services with hard guarantees from cycle harvesting resources. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID03)*, Tokyo, Japan, 2003. Available http://www.zurich.ibm.com/pdf/GridEconomics/HSQ_final.pdf.

[50] J. M. Kleinberg. Authoratitive sources in a hyperlinked environment. *Journal of the ACM*, 48:604–632, 1999. Available http://www.cs.cornell.edu/home/kleinber/auth.pdf.

[51] U. Kruschwitz and R. Sutcliffe. Query log analysis, 2007. http://http://cswww.essex.ac.uk/LAC/lcday6_papers/talk_UK.pdf.

[52] R. R. Larson and R. Sanderson. Grid-based digital libraries: Cheshire3 and distributed retrieval. In *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries*, Denver, Colorado, USA, July 2005. Available http://cheshire.berkeley.edu/sp245-Larson.pdf.

[53] J. H. Lee. Properties of extended boolean models in information retrieval. In *Proceedings of the 17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, page 182, Dublin, Ireland, July 1994. ACM. Available http://widit.slis.indiana.edu/irpub/SIGIR/1994/pdf19.pdf.

[54] H. Li, Y. Cao, J. Xu, Y. Hu, S. Li, and D. Meyerzon. A new approach to intranet search based on information extraction. In O. Herzog, H.-J. Schek, N. Fuhr, A. Chowdhury, and W. Teiken, editors, *CIKM*, pages 460–468, Bremen, Germany, October/November 2005. ACM. Available http://doi.acm.org/10.1145/1099554.1099685.

[55] M. Litzkow and M. Livny. Experience with the condor distributed batch system. In *Proceedings of the IEEE Workshop on Experimental Distributed Systems*, pages 97–101, Huntsville, Alabama, USA, October 1990.

[56] Lucene. Lucence search engine. Website, 2007. http://lucene.apache.org/.

[57] P. Lyman and H. R. Varian. How much information? *The Journal of Electronic Publishing*, 6(2), 2000. Available http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/.

[58] C. Manning, P.Raghavan, and H. Schütze. Introduction to information retrieval. Cambridge University Press. (To appear in 2008), 2008.

[59] O. A. McBryan. Genvl and wwww: Tools for taming the web. In O. Nierstarsz, editor, *Proceedings of the First International World Wide Web Conference*, pages 79–90, CERN, Geneva, 1994. Available http://www94.web.cern.ch/WWW94/PapersWWW94/mcbryan.ps.

[60] E. Meij and M. Rijke. Deploying Lucene on the grid. In *Open Source Information Retrieval Workshop at the 29th ACM SIGIR Conference on Research and Development on Information Retrieval*, Seattle, Washington, USA, August 2006. Available http://www.emse.fr/OSIR06/2006-osir-p25-meij.pdf.

[61] D. A. Menascé and E. Casalicchio. Quality of service aspects and metrics in grid computing. In *Int. CMG Conference*, pages 521–532. Computer Measurement Group, 2004. Available http://cs.gmu.edu/~menasce/papers/menasce-cmg-ny2005.pdf.

[62] Microsoft. Microsoft sharepoint. Website, 2008. http://www.microsoft.com/sharepoint/.

[63] A. Moffat and T. C. Bell. In-situ generation of compressed inverted files. *Journal of the American Society of Information Science*, 46(7):537–550, 1995.

[64] A. Moffat, W. Webber, and J. Zobel. Load balancing for term-distributed parallel retrieval. In E. N. Efthimiadis, S. T. Dumais, D. Hawking, and K. Järvelin, editors, *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 348–355, Seattle, Washington, USA, August 2006. ACM. Available http://doi.acm.org/10.1145/1148170.1148232.

[65] Netjmc. Highlights from the 2007 global intranet survey reports. Website, 2009. http://netjmc.typepad.com/globally_local/2007/11/highlights-from.html.

[66] G. B. Newby and K. Gamiel. Secure information sharing and information retrieval infrastructure with gridIR. In *Intelligence and Security Informatics, First NSF/NIJ Symposium*, volume 2665 of *Lecture Notes in Computer Science*, page 389. Springer, June 2003. Available http://www.gir-wg.org/papers.html.

[67] NHS. NHS Choices. Website, 2009. http://www.nhs.uk.

[68] OmniFind. OmniFind search engine. Website, 2008. http://www-306.ibm.com/software/data/enterprise-search/omnifind-yahoo.

[69] Open Grid Forum. The open grid forum. Website, 2007. http://www.ogf.org/.

[70] pdfbox.org. PDFBox - Java PDF ;ibrary. Website, 2007. www.pdfbox.org.

[71] M. Porter. Porter's stemming algorithm. Website, 2007. http://tartarus.org/~martin/index.html.

[72] J. A. Pouwelse, P. Garbacki, D. H. J. Epema, and H. J. Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In *4th International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 205–216, Ithaca, NY, USA, 2005. Available http://www.pds.twi.tudelft.nl/~pouwelse/CPE-Tribler-final.pdf.

[73] A. Rajasekar, M. Wan, R. Moore, W. Schroeder, G. Kremenek, A. Jagatheesan, C. Cowart, B. Zhu, S. Chen, and R. Olschanowsky. Storage resource broker - managing distributed data in a grid. *Computer Society of India Journal*, 33(4):45–28, October 2003.

[74] Y. Rasolofo, F. Abbaci, and J. Savoy. Approaches to collection selection and results merging for distributed information retrieval. In *CIKM*, pages 191–198, Atlanta, Georgia, USA, November 2001. ACM. Available http://doi.acm.org/10.1145/502585.502618.

[75] B. A. Ribeiro-Neto and R. A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *Proceedings of the 3rd ACM conference on digital libraries*, Pittsburgh, 1998. ACM Press, New York. Available http://doi.acm.org/10.1145/276675.276695.

[76] K. Risvik and R. Michelsen. Search engines and web dynamics. *Computer Networks*, 39(3):289–302, 2002. Available http://www.idi.ntnu.no/~algkon/generelt/se-dynamicweb1.pdf.

[77] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11), 1975. Available http://doi.acm.org/10.1145/361219.361220.

[78] R. Sanderson and R. R. Larson. Indexing and searching tera-scale grid-based digital libraries. In X. Jia, editor, *Infoscale*, volume 152, page 3, Hong Kong, 2006. ACM. Available http://doi.acm.org/10.1145/1146847.1146850.

[79] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 222–229, Tampere, Finland, July 2002. ACM. Available http://doi.acm.org/10.1145/564376.564416.

[80] F. Scholze, G. Haya, J. Vigen, and P. Prazak. Project grace - a grid based search tool for the global digital library. In *Proceedings of the 7th International Conference on Electronic Theses and Dissertations*, Lexington, Kentucky, USA, June 2004.

[81] SETI@Home. Search for extraterrestrial intelligence at home. Website, 2007. http://setiathome.berkeley.edu/.

[82] W.-Y. Shieh and C.-P. Chung. A statistics-based approach to incrementally update inverted files. *Information Processing and Management*, 41(2):275–288, 2005. Available http://dx.doi.org/10.1016/j.ipm.2003.10.004.

[83] SRB. The SRB datagrid. Website, 2007. http://www.sdsc.edu/srb.

[84] H. Sutter. The free lunch is over a fundamental turn toward concurrency in software. *C/C++ Users Journal*, 23(2), February 2005. Available http://www.gotw.ca/publications/concurrency-ddj.htm.

[85] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor - a distributed job scheduler. In T. Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, 2001. Available http://www.cs.wisc.edu/condor/doc/beowulf-chapter-rev1.ps.

[86] Technorati Inc. Technorati Media. Website, 2009. http://technoratimedia.com.

[87] textmining.org. TextMining.org. Website, 2007. www.textmining.org.

[88] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4), 2005. Available http://www.cs.wisc.edu/condor/doc/condor-practice.pdf.

[89] The Apache Software Foundation. Apache Tomcat. Website, 2007. http://tomcat.apache.org/.

[90] The Apache Software Foundation. WebServices - Axis. Website, 2007. http://ws.apache.org/axis/.

[91] A. Tomasic and H. Garcia-Molina. Query processing and inverted indices in shared-nothing document information retrieval systems. *The VLDB Journal*, 2(3):243–275, July 1993. Available http://www-db.stanford.edu/pub/tomasic/1993/pdis.93.ps.

[92] A. Tomasic, H. Garcia-Molina, and K. Shoens. Incremental updates of inverted lists for text document retrieval. In *Proceedings of the 1994 ACM SIGMOD Conference on Management of Data*, pages 289–300, Minneapolis, Minnesota, USA, 1994. Available http://doi.acm.org/10.1145/191839.191896.

[93] E. M. Voorhees, N. K. Gupta, and B. Johnson-Laird. Learning collection fusion strategies. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 172–179, Seattle, Washington, USA, July 1995. Available http://doi.acm.org/10.1145/215206.215357.

[94] Y. Wang. GridFTP and GridRPC. Website, 2008. University of Maryland, Department of Computer Science. http://www.cs.umd.edu/class/spring2004/cmsc818s/Lectures/gridftp-rpc.pdf.

[95] I. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, California, USA, 1999.

[96] D. A. Wood and M. D. Hill. Cost-effective parallel computing. *COMPUTER: IEEE Computer*, 28:69–72, 1995. Available ftp://ftp.cs.wisc.edu/wwt/computer95_cost.pdf.

[97] W. Xi, O. Sornil, M. Luo, and E. A. Fox. Hybrid partition inverted files: Experimental validation. pages 422–431, 2002.

[98] J. Xu and W. B. Croft. Query expansion using local and global document analysis. In E. N. Efthimiadis, S. T. Dumais, D. Hawking, and K. Järvelin, editors, *Proceedings of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 4–11, Zurich, Switzerland, August 1996. Available http://doi.acm.org/10.1145/243199.243202.

Appendix A

Queries used in the Experiments

1901 census	2008 olympic games	2012	2012 london	2012 olympics
6 nations	abuse	abused	acas	acids
adsl	adt	age of consent	agent orange	alarm
alarm system	alarm systems	alarms	aldrich	amazon
amazon uk	ambulance	ambulance service	american embassy	american embassy london
ana	ana anorexia	anatomy	and1	anger management
angles	anorexia	anorexia nervosa	anorexic	anpe
antenna	aol	ares	arkadin	armistice
armistice day	arrest	arsenal	art	artists
ashes	ask frank	assault	asterisk	astronomy
athens 2004	autosport	avian flu	avon police	avon somerset police
azureus	bacp	bar directory	bar exam	basket
basket ball	basketball	basketball games	bathroom scales	baton
battle	bbc cricket	bbc rugby	bbc sport	bearshare
becca	beijing	beijing 2008	beijing olympic	beijing olympics
beth tweddle	big bang	billabong	biodiversity	bird flu
bit torrent	bittorrent	black book	bloods	bobby
body	body image	bones	books	boys food
bp	bps	braces	brain	brands hatch
bring it on	brita	britain	british cheerleading	british crime survey
british embassy	british gas	british gymnastics	british legion	broadband
broadband bt	broadband uk	bt	bt internet	bt line
bt login	bt mail	bt yahoo	btcc	btinternet
btyahoo	bulimia	burton	cafcass	callaway
canoe	car racing	carbon	carbon dioxide	carbon emissions
carbon footprint	carbon neutral	carbon trust	cas	ccm
cencus	census	census 2001	census records	census uk
cern	champions league	championship	change name	changing name
cheer	cheerleader	cheerleaders	cheerleading	cheerleading bca
cheerleading outfits	cheerleading uniforms	cheers	chelsea	chemical
chemistry	chicken pox	chickenpox	child abuse	child support
children act	children act 1989	childrens act	chloride	cholera
chromatography	chubb	churchill	cia	circle
civil service	civil war	claro	claro ideias	climate change
co2	co2 emissions	cognitive	cold	cold flu
colds	college law	college of law	comet	common cold
compounds	conference call	conference calls	conferencing	constructive dismissal
contagious	contaminacion	copyright	copyright law	copyright laws
copyright uk	cough	counseling	counselling	countries
county cricket	courier	crb	crb check	crb checks
cricinfo	cricket	cricket bats	cricket game	cricket games
cricket score	cricket scores	cricket world	crime	crime library
crime statistics	crimestoppers	crimewatch	criminal check	criminal record
criminal records	criminal records bureau	crips	cruse	custody
cycle life	dancing on ice	death row	dect	demand
dental practice	dental surgery	dentist	dentistry	dentists
dgi	dhl	digestive system	digital	digital scales
digital tv	disclosure	discrimination	dish	dive
diving	divorce	divorce law	donnington	dow
download	download limewire	download skype	dream	dream dictionary

dream interpretation	dream meanings	dreams	dreams meaning	dti
dunk	dwyco	east hockey	eating disorder	eating disorders
ebola	eclipse	ecological	ecology	economic
economic growth	economics	economy	ecosystem	edf
einstein	electric fires	electricity	electron	electron microscope
element	elements	elite law college	ellis	ellis brigham
embassy	embassy london	employee rights	employer	employment
employment law	employment tribunal	employment tribunals	emule	energy
enfermedades	england basketball	england cricket	england rugby	equation
erosion	european patent	european tour	eurostar	everton
every child matters	executor	f1	fa	family law
fantasy football	fathers	fax	fbi	fbi most wanted
field hockey	fifa	fire	fire and rescue	fire brigade
fire of london	fire service	fireman	fireman sam	fireplace
fireplaces	fires	fisher scientific	flora	flu
flu symptoms	food chain	food chains	food web	football
forces	fordham	forensic	forensic science	formula 1
formula one	fractions	frank	frank drugs	fred west
free limewire	freesat	freeview	freeview box	freeview tv
french embassy	freud	friction	future cheer	gang
gangs	gangsters	gas	gas fire	gas fires
gases	gdp	geology	german embassy	gimnasia
gizmo	glandular fever	globalisation	glue	golf
golf club	golf clubs	golf course	golf courses	goodwood
gotomeeting	grand prix	graph	graphs	gravity
grays	great fire london	gsf	guildford college	gumball
gums	gunge	gymnast	gymnastic	gymnastic leotards
gymnastics	gymnastics leotards	gymnasts	hadron	hadron collider
handwriting	hanna	harassment	harlequins	harry potter
harvard law	harvard law school	history	hockey	hockey england
hockey equipment	hockey stick	hockey sticks	home security	homeless
homeless people	homelessness	hughes	human body	human skeleton
hydrochloric acid	ice hockey	ice rink	ice skating	icom
igneous	impact	income	india cricket	inflation
inflation rate	influenza	inghams	injection moulding	intellectual property
intercall	internet phone	ionic	ip phone	iplayer
irons	isaac newton	isohunt	isp	italian embassy
iverson	jail	jealous	joints	karting
kazaa	kemp	kenneth	kitchen scales	knicks
kobe	labview	lakers	languages	large hadron collider
lavinia	law school	law schools	le mans	leafs
legion	legionella	leotard	leotards	lewis hamilton
lhc	lime wire	limestone	limewire	live cricket
liver	liverpool	liverpool fc	locks	locksmith
locksmiths	loctite	london economics	london school economics	london theatre
lpc	lpg	lsat	lse	mafia
maglite	magnets	malaria	malaria tablets	man utd
manchester united	manns	marie curie	mars	maslow
math	math games	mathematics	maths	maths games
mda	mean	medieval	meningitis	meridian
meteor	mia ana	michael jordan	microscope	microscopes
milano leotards	mini golf	mini pool	mini pool 2	mininova
minipool	mitel	mizuno	momentum	monetary
moon	moonpig	mortality	most wanted	motorsport
mount st helens	mountain	mountains	mouth	mouth ulcers
move on	msds	mt st helens	mucus	multiplication
muscle	muscles	myers briggs	nasa	national archives
national statistics	nba	nba basketball	nervosa	nervous system
new york bar	newton	nhl	nhs dental	nhs dentist
nhs dentists	night vision	nitrate	nlp	noel edmonds
not	npower	ntl	nuclear	o2 shop
oecd	office national statistics	oil	olympic 2008	olympic 2012
olympic games	olympic games beijing	olympics	olympics 2008	olympics bbc
olympics london	online sector	open golf	orange broadband	organs
ospreys	oyster	pace	padi	pakistan cricket
palin	pandemic	paramedic	parcel	parcelforce
pasteur	patent	patent office	patent office uk	patent search
patent uk	patents	penal	performing rights	periodic table
personality	personality test	petrol	pga	pga golf
pga tour	ph	phlegm	physics	pi
piaget	ping	pirate bay	pitney bowes	plagiarism
planet	planets	plastic	plastics	play
playing with fire	pluto	poem	poems	poetry
police	police jobs	police officer	police training	police uk
polycom	polymer	polymers	pom	pom pom
pom poms	pompeii	poms	poms poms	pools
poppy	poppy appeal	poppy day	poppy day appeal	poppy remembrance
population	population statistics	post	post code finder	post codes
post office	postal address	postcode	postcode finder	postcodes
potassium	poverty	powell	powergen	powwownow
pox	pregnancy chicken pox	premier league	premiership	prison
prisons	pro ana	pro anorexia	pro mia	pro-ana

probate	probate registry	prs	psychologist	psychology
psychotherapy	qos	quantum	quicksilver	railway
rally	rangers	rapidshare	rate of inflation	rate of reaction
recession	recycle	recycling	redundancy	redundancy calculator
redundancy pay	redundancy payments	regression	rehabilitation of offenders	relationships
rememberance	rememberance day	remembrance	remembrance day	remembrance day poppy
remembrance sunday	resin	resource centre	respiratory system	rfu
rhythmic	rico	rocks	roman	roman numerals
roxy	royal british legion	royal legion	royal mail	royal mail delivery
royal mail post	rubber	rugby	rugby 2007	rugby fixtures
rugby league	rugby tickets	rugby union	rugby world	rugby world cup
russian embassy	ryder cup	safe	safes	salter
salter scales	salto	satellite	scales	school of law
schumacher	scottish power	scuba	scuba diving	security
security alarms	security systems	sedimentary	separation	serial killer
serial killers	sexual abuse	shakespeare	shapes	shawn johnson
sheriff	shingles	shingles symptoms	shipping	shipwrecked
sightspeed	sigma	sigma aldrich	silverstone	simply travel
sinus	sinuses	sinusitis	sip	sipgate
six nations	size zero	skating	skeletal	skeleton
ski	ski deals	ski france	ski holiday	ski holidays
ski resorts	ski wear	skiing	skiing holidays	skinner
skinny	skipe	skis	skpe	sky
sky box	sky broadband	sky channels	sky digital	sky hd
sky plus	sky remote	sky sports	sky tv	skyp
skype	skype phone	skype phones	snow and rock	snow rock
snowboard	snowboarding	social benefits	social policy	social security
social security number	social security office	social security offices	social security uk	social welfare
sodium	soil	solar	solar system	somerset police
sore throat	sorting office	southern electric	spanish embassy	spectra
speedo	speedway	sputnik	st andrews	st helens
stagecoach	standard deviation	stars	statistics	statutory redundancy
stick cricket	stickcricket	sticks	storage	strathclyde
streetball	strep throat	suicide	surf	surfboards
swat	swim	swimming	swimming baths	swimming pool
swimming pools	swimwear	talk frank	talk to frank	tamiflu
tandberg	tanita	taylor made	tectonic	teeth
teleconference	teleconferencing	telescope	telescopes	test cricket
test match	thames water	the body	the brain	the earth
the embassy	the human body	the law college	the moon	the olympics
the periodic table	the planets	the skeleton	the solar system	the welfare state
theatre royal	theatre tickets	therapist	thermo	thin
thinspiration	throat	throat infection	tiger woods	timeline
tiscali	tiscali broadband	titanic	tongue	tonsilitis
tonsilitus	tonsillitis	tooth	toothbrush	toothpaste
torch	torches	torrent	torrents	torrentspy
tottenham	trade mark	trade marks	trademark	trademarks
trains	trampolining	transmitter	transport	treaty
trent fm	tribunal	trusts	tudor	tv
twickenham	ufficio	uk cheerleading	uk divorce	uk embassy
uk employment law	uk fire	uk gymnastics	uk population	uk post
uk post office	uk statistics	uk unemployment	ukca	ulcers
unemployment	unemployment benefit	unemployment benefits	unemployment insurance	unemployment office
unfair dismissal	united nations	universe	university law	university of law
us embassy	us embassy london	us patent	usa embassy	utilities
utorrent	van	vesuvius	veteran	veterans
victorian	video conference	video conferencing	videoconferencing	virgin broadband
voda	vodafone	vodafone customer	vodafone customer services	vodafone ireland
vodafone live	vodafone topup	vodafone uk	vodaphone	vodophone
voip	voip phone	voipbuster	voipcheap	voipstunt
volcano	volcanoes	volcanos	vonage	vwr
wales rugby	war	wasps	waste	waterstones,
wave	wavelength	waves	weathering	web conferencing
webex	webinar	webx	weighing	weighing scales
welfare	welfare policy	welfare state	welsh rugby	west ham
whitening teeth	wicked weasel	wills	wind	wisdom teeth
woods	workplace	worksheets	world cup	world cup cricket
world hockey	world population	world war	wrc	wru
ww1	ww2	yahoo	yale	young leafs
youngleafs	youth crime	zone leotards		