A DYNAMIC QUERY/INDEX ROLE
CLUSTER BASED SEARCH ENGINE

By: NDAPANDULA NAKASHOLE
Supervisor: Dr. Hussein Suleman

This report is submitted in partial fulfilment of
the requirements for the degree of
Bachelor of Science in Computer Science (Honours)

Department of Computer Science
University of Cape Town
South Africa

November 2006
Abstract

The Internet has become a vast information resource in recent years. To help users find desired data in this environment, many powerful search engines have been created. However, these search engines are not flawless; one common problem among search engines is that of returning outdated results which may be regarded erroneous in some contexts. The reason for this discrepancy lies in the way various search engines perform indexing. Indexing is often carried out on a discrete basis, with the time intervals spanning weeks or months depending on the search engine. However, some websites, for example news sites, change frequently and will not work with a search facility whose index is updated on a monthly basis.

While some current large search engine organizations have developed solutions that help minimize outdated results, little work has been done to develop a flexible and cost-effective solution that can work on a relatively small cluster of workstations. This project proposes to design and implement a search engine which efficiently utilizes a cluster with a small number of workstations with the goal of minimizing outdated results. The project was split into two clearly separable parts: the indexing module, which the author was responsible for and is documented in this report; and the querying module, which the author’s project partner worked on. The software developed dynamically allocates cluster nodes to the roles of indexing and querying based on load averages of the individual nodes. The system enables efficient system resource utilization by allocating resources where they are most needed. Experiments were carried out to compare the performance of dynamic allocation versus static allocation. Different data sizes were used for experiments, with the largest one having just over 100,000 files. Experimental results enable visual interpretation and comparison of different scenarios. Possible extensions include testing with larger data sets and incorporating fault-tolerance.
Table of Contents

List of Figures iv
List of Tables iv

1. Introduction and Motivation 1
   1.1 Importance of the Project 2
   1.2 Project Objectives 2
   1.3 Report Outline 3

2. Problem Definition 4
   2.1 Problem of Outdated Results 4
   2.2 Research Question Addressed 6
   2.3 Project Requirements 6
      2.3.1 Overall system requirements 6
      2.3.2 Indexing-specific requirements 6
   2.4 Project Scope 7

3. Background 8
   3.1 Theory 8
      3.1.1 Search engine concepts 8
      3.1.2 Cluster computing 10
   3.2 Related Work 11
      3.2.1 Crawlers 11
      3.2.2 Inverted index optimizations 12
      3.2.3 Search engine cluster architectures 13

4. Design 15
   4.1 Overview of the Solution 15
   4.2 Features of the Indexing System Solution 16
      4.2.1 Inverted indices 16
      4.2.2 Fresh indices 16
      4.2.3 Load balancing and consistency 17
      4.2.4 Scalability 17
      4.2.5 Cluster utilization 17
   4.3 Overall Indexing System Design 18
      4.3.1 The structure of the Index 19
      4.3.2 The ID-URL index 19
   4.4 Algorithms 20
      4.4.1 Master-slave approach and algorithm 20
      4.4.2 The indexing algorithm 25
      4.4.3 The index update algorithm 25
   4.5 The Parser 27
   4.6 Package Diagrams 27
   4.7 Class Diagrams 28
## 5. Experimental Design  
5.1 Basis of the Hypothesis  
5.2 The Hypothesis  
5.3 The Experiments  
5.4 Equipment

## 6. Results  
6.1 Index Creating Program  
   6.1.1 Reallocation interval  
   6.2.2 Dynamic versus static allocation  
   6.2.3 System Resource Utilization  
6.2 Index Updating Program  
6.3 Discussion  
   6.3.1 Index creating  
   6.3.2 Index updating  
   6.3.3 Scope and Limitations of Results

## 7. Conclusions

## 8. Future Work  
8.1 Fault Tolerance  
8.2 Index Compression  
8.3 A General-Purpose Parser  
8.4 Load-Balancing  
8.5 Document Summaries  
8.6 Index Update Algorithm

## 9. References

Appendix
List of Figures

Figure 1: The search engine reference model. Redrawn from [3] 4
Figure 2: Example of a heterogeneous crawler. Adapted from [3] 5
Figure 3: Core Components of a search engine. 8
Figure 4: Typical cluster computer architecture 10
Figure 5: The design of the entire search engine system 15
Figure 6: The core components of the Indexing system 18
Figure 7: The master-slave model 20
Figure 8: An outline of the master-slave algorithm 21
Figure 9: The operation carried out by slave processors to ensure ID uniqueness 25
Figure 10: An outline of the indexing algorithm 25
Figure 11: An outline of the index update algorithm 26
Figure 12: Package diagram for the indexing module 27
Figure 13: Class diagram for the indexing system 29
Figure 14: Effect of reallocation interval on the time to index 119 423 files 33
Figure 15: Effect of reallocation interval on the time to index 50149 files 34
Figure 16: Effect of reallocation interval on the time to index 11475 files 35
Figure 17: Performance of static and dynamic allocation 35
Figure 18: Allocation of machines based load average ratio 36
Figure 19: Number of idle machines with dynamic and static allocation 37
Figure 20: Time to update versus data size 38

List of Tables

Table 1: Files and directories created by the master program 23
Table 2: Auxiliary files and directories created by slave processors 24
Table 3: Description of the packages that make up the system 28
Table 4: Description of classes belonging to the Indexing system 28
Table 5: Data used to draw graph in Figure 14 46
Table 6: Data used to draw graph in Figure 15 46
Table 7: Data used to draw graph in Figure 16 47
Table 8: Data used to draw graph in Figure 17 47
Table 9: Data used to draw graph in Figure 18 47
Table 10: Data used to draw graph in Figure 20 47
1. Introduction and Motivation

There has been a vast increase in the amount of information available for human consumption in recent years. Dynamic environments such as the Web are growing exponentially as more and more information is made available. As a result, it is difficult to find information of interest in such environments. For this reason, search engines have grown into the most popular way for navigating large amounts of information.

The dynamic nature of the Web presents challenges to search engine design. The reason for this is that search engines need to keep a local store of the webpages that need to be indexed. However, because the Web changes frequently, it becomes difficult to keep the local store synchronized with the Web. If the local store is not updated often enough, results returned to users will be outdated. Most search engines update their indices on a discrete basis, with time intervals spanning a few days or weeks. This is reasonable for an average webpage as research has shown that once a page is created it either goes through minor changes or no changes at all [2]. However, a search facility whose index is updated on a monthly basis will not produce the most up-to-date results for websites that change frequently.

Search engines need to index very large amounts of data, while maintaining fast response times to user queries. These requirements necessitate high performance computing. This is where parallel computing fits in; clusters in particular have the desirable features of high performance, scalability and fault-tolerance. For this reason, clusters are the architecture on which the majority of existing search engines are based. Furthermore, clusters have better price to performance ratios than alternative high performance computers [1].

In an attempt to minimize outdated results returned to users, some large organizations came up with a solution that uses specialized crawlers, whereby there are a number of nodes in the cluster dedicated to crawling dynamic content as often as possible. This is a viable solution as can be seen in the working example of the Google search engine [4]. However, this solution is not flexible as it cannot be effectively deployed on a small cluster consisting of about 10 machines.

This project proposes, implements, and tests a cluster-based search engine which efficiently utilizes a cluster with a relatively small number of workstations with the goal of minimizing outdated results. More specifically, the project explores the feasibility of dynamically allocating the roles of indexing and querying to nodes in a cluster, depending on the load of the system. The aim is to index updated data as soon as possible while not compromising response times to queries.

The software that was developed monitors the system load in order to optimize the proportion of nodes that are allocated to the roles of indexing and querying. If, for example, the system receives a large amount of data to be indexed, then a large proportion of nodes will be allocated to indexing. On the other hand if the system receives a lot of queries at one time (also known as the flash crowd effect), then a large
proportion of nodes in the cluster will be allocated to serve incoming queries. The system was implemented in C++ in conjunction with the MPI library for parallel programming. The C++ programming language was chosen over Java particularly because C++ has well-established parallel programming libraries. Furthermore, C++ execution speeds are suitable for high performance computing. MPI was favoured over other parallel programming tools for its portability and for the fact that it is a formal specification which is widely used. The choice of the programming language and tools was influenced by the time available for the project.

1.1 Importance of the Project

This project has potential effects on various search engine application areas. The most pertinent of these areas are documented below.

Hospitals often need to access patients records from all national hospitals that a patient has been to. A search facility with the ability to synchronize data across all hospitals in a country provides the desired functionality to health professionals. However, for this to work well, results returned need to be as current as possible so that the right treatment is given based on the correct medical history. In this case the search facility needs to maintain relatively fast response times to queries while at the same time keeping patient records up-to-date. Once again, in developing nations there are not enough computing resources required to employ a resource intensive solution. The flexible solution developed in this project makes it an attractive option in this case. Furthermore, the idea behind this project is highly applicable in this scenario. Cluster machines will be allocated to the roles of fetching data and handling user queries, depending on the load of the system.

Small companies are not in a financial position to acquire the computing power required by the solution used by large scale search engine organizations. Thus they need a means of producing up-to-date results with the amount of computing power at their disposal. Currently in such small companies there is no means of ensuring that results returned to users are up-to-date. This often results in outdated results and hence unhappy users. Users are then left with no choice but to resort to other means of obtaining information of interest. This project provides a flexible solution that can be used by such small companies. This will help in minimizing outdated results returned to users.

1.2 Project Objectives

In terms of project deliverables, the project objectives were separated into two clearly separable parts:

1. The Query Module. This part handles the querying aspects of the search engine. The author’s project partner, Calvin Pedzai, was responsible for this part of the project; further details on this can be found in his report.
2. The Indexing module. This was the author’s part of the project; this report details the design, implementation, testing and evaluation of this module. A detailed discussion about the project requirements and objectives concerning the indexing module can be found in Chapter 2.
1.3 Report Outline
The rest of this report follows the following structure to describe various stages of the project life-cycle. Chapter 2 details the problem definition including the project requirements and how these were elicited. Chapter 3 discusses related work and underlying theory in the fields of search engine design and cluster computing in order to put the problem space into perspective. The design of the project’s software product is discussed in Chapter 4. Chapter 5 and 6 document the experimental design and results respectively. Chapter 7 has some concluding remarks. Finally, Chapter 8 presents possible areas for future work that could be done to expand on the indexing module.
2. Problem Definition

This chapter describes the problem which this system aimed to solve. The chapter details the project requirements and explains how they are elicited from the problem definition. Furthermore, the research questions that the project aimed to tackle are presented.

2.1 Problem of Outdated Results

Internet-scale search engines have to deal with vast amounts of data. At the same time they have to respond to thousands of queries simultaneously. Thus for such search engines to provide quick responses to user queries, the pages for which search capabilities are provided are usually fetched from individual Web servers by the crawler and stored in a local store for later processing by the indexing module (Refer to the diagram below for high level search engine reference model). In order to return up-to-date results which reflect the current state of the Web, it is essential that the local store be kept as fresh as possible.

![Figure 1: The search engine reference model. Redrawn from [3]](image)

The importance of up-to-date results is one that cannot be overlooked. The impact of outdated results can range from undesirable to catastrophic depending on the context. Consider an online shop that sells goods online. A user at his/her terminal issues a query for a certain item that is out of stock, however the search returns that the item is in fact available. The user then proceeds to purchase the item. This has an undesirable consequence as the shop has now sold one item to more than one customer; this creates numerous problems.
Several successful attempts have been made to ensure that search engines return correct and up-to-date results. Many organizations established that search engines need massive computational power in order to handle the large amounts of data they have to deal with. In a description of the Google cluster architecture, Borroso, Dean and Holzle [5] admit this need as follows: “Few Web services require as much computation per request as search engines.” Owing to this need, many large organizations have looked at increasing their cluster sizes. Google’s architecture features clusters of more than 15,000 commodity PCs [5]. With such an amount of computing power available to them, large organizations have taken advantage of this computational power to provide fresh results.

In addition to using a scheduling algorithm to refresh its local store, the FAST search engine [3] uses heterogeneous crawlers as a supplement. This handles the fact that having a fresh copy of some documents in the repository is not as valuable as having a fresh copy of some other documents. This is true for dynamic content, for example news. This type of content must be refreshed and indexed very often to be of any value to users. By realizing that a scheduling algorithm alone cannot ensure that dynamic content is given preference when refreshing documents, the FAST search engine provides this functionality through heterogeneous crawlers. The diagram below shows a heterogeneous crawler deployment employed by the FAST search engine. The FAST search engine uses separate clusters for crawlers, thus what is shown in Figure 2 is an example of a crawler cluster.

![Figure 2: Example of a heterogeneous crawler. Adapted from [3]](image)

The crawler cluster shown in Figure 2 has a few dedicated machines for crawling news content in addition to the large-scale web crawler. In the example, some of the crawler cluster machines make up the multimedia crawler. From the description of the solution and by looking at the Figure above, one can see that it is not feasible to apply heterogeneity as a solution in a small cluster. Sub-dividing the already small cluster to dedicate machines to crawl certain types of content will potentially have a negative impact on the overall system performance. This solution is thus not flexible as a small company with a cluster of about 10 nodes will not be able to utilize it.
Indexing and querying are considered as roles that have to be executed separately on different machines for the following reasons; firstly, it is important to separate these roles on different machines in order for the system to be robust. If machines are overloaded with both responding to queries and indexing, they could crash and result in a complete system failure; secondly, this separation is necessary for consistency reasons, for example, it is undesirable for both indexing and querying to be accessing the same files as it could lead to various errors.

2.2 Research Question Addressed
The main research question which the indexing part of the project addressed can be stated as follows:

"Is it possible to create an indexing module on a relatively small cluster, which indexes updated data as soon as possible and utilises system resources effectively under different work loads?"

This research question focuses on using minimal computational resources while producing fresh results, which is the ultimate aim of this project. The question has to parts to it; the fist part is concerned with the ability to maintain fresh indices by providing fast indexing and hence indexing updated data as soon as possible; the second part is concerned with allocating system resources where they are most needed. The two parts of the research question are equally important. The answer to the research question will give a clear indication of the success of the developed system.

2.3 Project Requirements
This section describes the project requirements which are elicited from the problem definition and research questions. The section gives the overall requirements of the project as a whole and the specific requirements for the Indexing module.

2.3.1 Overall system requirements
The high level system requirements are a search engine that is:
2. Capable of dynamically allocating the roles of indexing and querying according to the system load.
3. Able to utilize system resources efficiently under different use-case scenarios.
4. Scalable.

2.3.2 Indexing-specific requirements
The specific requirements for the Indexing module are to design an indexing module that:
1. Runs on a cluster.
2. Maintains fresh indices.
3. Uses an efficient index maintenance strategy
4. Scales with increasing size of data to be indexed.
2.4 Project Scope

The purpose of this section is to narrow down the project domain, in order to make the project feasible in the time allocated to the project. Since the focus of the project is not on developing new crawling strategies and algorithms but rather on efficient utilization of a cluster, a new crawler was not developed. An existing crawler that meets the project needs was used for this purpose. In addition, although testing with large and varied amounts of data would be good for determining the scalability of the Indexing module, only pages from the uct.ac.za domain were used as test data. The reason for this was to allow adequate testing and analysis of the results in the time allotted to the project. Finally, due to the fact that the project period is limited to a short time, no complex parallel algorithms were used for load balancing among the nodes allocated to indexing, but rather a self-scheduling master-slave algorithm was employed.
3. Background

This chapter details the theoretical background on search engine and cluster computing concepts that are necessary to understand the developed solution. The chapter further presents related work in the areas of indexing and cluster architectures with particular emphasize on literature related to maintaining fresh indices.

3.1 Theory

This section describes the basic concepts of search engine design and cluster computing. The section serves to provide an overview of how search engines work and to provide a foundation for the solution described in Chapter 4 of this report.

3.1.1 Search engine concepts

There are a few components which are common to all search engines. Knowledge of these components is necessary to understand the fundamental search engine architecture. Figure 3 below depicts the general architecture of search engines; each component is discussed further in this section.

![Figure 3: Core Components of a search engine.](image)

**The Crawler**

A crawler is a program that browses the Web by following URLs. The program is provided with a set of URLs whose pages it retrieves from the Web. The crawler places the initial set of URLs in a queue where all URLs to be retrieved are kept and prioritized. From this queue, the crawler gets a URL (in some order), downloads the page, extracts...
any URLs in the downloaded page, and puts the new URLs in the queue. The crawler continuously downloads pages from individual servers until local resources, such as storage, are exhausted or the queue is emptied [8]. The crawler passes the retrieved pages into a local store which is a snapshot of a subset of the Web.

**Local Store**
The local store in Figure 3.1 is for managing collections of Web pages. The local store needs to perform two basic functions. First, it must provide an interface for the crawler to store pages. Second, it must provide an efficient access mechanism that the indexer can use to retrieve pages [8].

**Indexer Module**
The role of the indexer is essentially to record which words appear in each page. For each encountered word, the indexing system maintains a set of URLs the word is relevant to, possibly along with other positional information regarding the individual occurrences of words. These indices must be kept in a format that allows their fast intersection and merging during querying time [9]. Thus the index is normally stored as inverted files. The inverted file for a term is a list of locations where the term appears in the collection [8]. In the simplest case, a location consists of a page identifier and the position of the term in the page. Below is an example of a simple inverted index.

<table>
<thead>
<tr>
<th>term</th>
<th>locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>7, 9 5, 10</td>
</tr>
<tr>
<td>dog</td>
<td>5, 2 3, 1</td>
</tr>
<tr>
<td>rat</td>
<td>6, 3</td>
</tr>
</tbody>
</table>

This means that the word cat appears in document 7, 9 times, and in document 5, 10 times. Similarly, the word dog appears in documents 5, and 3. The word rat is in document 6.

Indexing systems based on the Google architecture [4] also associate with a page and index some text that is related to or contained in the links pointing to the page. Such text is often referred to as anchor text. The indexing system can leverage the page descriptions embedded into its incoming links by making use of anchor text.

**The query module**
A query engine accepts search queries from users and performs searches on the indices. The query module ranks the results before returning them to the user, such that the results near the top are the most likely to be what the user is looking for. The query module also optionally generates a summary for each search result, often based on the Web page repository or local store. In some search engines, the query module is also responsible for caching the results of popular search queries.
3.1.2 Cluster computing

“There are three ways to do anything faster:
1. Work harder.
2. Work smarter.

In computer terms, to work harder is to enable the computer to do more work by increasing processor speed. Working smarter means that the programmer devises better and efficient algorithms to do the task faster. Getting help involves using multiple processors to work on the same task simultaneously; this is what parallel computing is about. The principle behind parallel computing is based on the fact that the process of solving a problem can be divided into subtasks which may be carried out simultaneously with necessary coordination.

A cluster is a type of parallel or distributed processing system that consists of a collection of interconnected stand-alone computers working together as a single, integrated computing resource [19]. A computer node can be a single or multiprocessor system with memory, I/O facilities and an operating system. A cluster generally refers to two or more computer nodes connected together. The nodes can exist in a single cabinet or be physically separated and connected via a LAN. An inter-connected cluster of computers can appear as a single system to users and applications. Such a system can provide a cost-effective way to gain fast and reliable services that have historically been found only on more expensive proprietary shared memory systems [19]. A typical architecture of a cluster is shown in Figure 4 below.

![Figure 4: Typical cluster computer architecture](image)
Some of the most pertinent components of a cluster computer are:

- Multiple High Performance Computers (PCs or Workstations)
- State-of-the-art Operating Systems (Layered or Micro-kernel based)
- High Performance Networks/Switches (such as Gigabit Ethernet and Myrinet)
- Parallel Programming Environments and Tools (such as parallel compilers, PVM and MPI)

The availability of standard programming tools and utilities has made clusters a practical alternative as a parallel-processing platform [19]. Such tools include message passing libraries and debuggers. Currently the most popular message passing libraries are the Parallel Virtual Machine (PVM) and the Message Passing Interface (MPI). PVM is both an environment and a message passing library, which can be used to run parallel applications on various systems such as supercomputers and clusters of workstations. MPI is a message passing specification, designed to be a standard for distributed memory parallel computing using explicit message passing.

3.2 Related Work

3.2.1 Crawlers

As Figure 3 illustrates, a crawler is responsible for downloading Web pages from the Web and providing a local store of these Web pages available for indexing. The page revisit policy of a crawler is an important factor in determining the freshness of the indices. A number of works have focused on finding answers to the questions that arise when attempting to develop a crawling strategy that maximizes the freshness of the local store [7, 10, 11 & 12]. Examples of these questions are: When are new pages crawled? When do pages get re-crawled? How are deleted pages dealt with?

The most common types of crawlers in literature are batch and incremental crawlers. This project employs an incremental crawler; this type of crawler never erases its local stores. On the other hand, a batch crawler starts from scratch in every indexing cycle. The FAST crawler [3], a crawler for the FAST search engine, is an incremental crawler. As an incremental crawler, at the beginning of each cycle, it begins where it left off when the last indexing phase stopped [3]. Although the FAST search engine caters for keeping a fresh local store, little is said about timely indexing so that the indexer is not outdone by the crawler. This project hopes to address this problem through dynamic allocation of nodes to indexing and querying. It is important that the indexer makes the index available for the query processor in a timely manner. If the indexer is very slow, a fresh local store will not affect the results that users get, hence the problem of outdated results will persist.

A crawler which is well documented is that employed by the Nutch search engine. Nutch is an open source implementation of a search engine, which is built on top of Lucene. Lucene is an API for text indexing and searching [21, 22]. The Nutch crawler fetches pages and turns them into an inverted index, which the searcher uses to answer users' search queries. It maintains several data structures including a set of segments and an inverted index. A Nutch segment is a collection of pages fetched by the crawler and is
indexed in a single run. At the end of every indexing cycle all the segments are merged to form one index that will be used for searching.

The Nutch crawler uses a uniform refresh policy where all pages are re-fetched at the same interval (30 days, by default) regardless of how frequently they change. This will not provide a useful search facility for frequently changing content. To leverage this, the Nutch crawler also provides functionality to set individual re-crawl deadlines on every page. While this can be utilized to provide a mechanism to ensure a fresh index, it is tedious as it has to be done per page.

As mentioned earlier, the main aim of this project is to investigate efficient use of a cluster and not to come up with a new crawling strategy; as such the project employs an existing crawler and document update model.

3.2.2 Inverted index optimizations

As mentioned earlier, search engines have demanding performance requirements. The quantity of data that must be searched in response to each query is very large. Inverted indices are employed by nearly all practical search engines to facilitate fast response times to queries. Thus query processing times on large amounts of data are dominated by the need to retrieve and scan the inverted file of each query term [16]. It is therefore important to optimize inverted indices in order to enhance performance. This sub-section reviews literature on efficient storage and maintenance of indices as index optimization strategies.

Index Compression

An inverted index consists of two major components: the terms from the data collection being indexed and inverted lists of document postings that store frequencies of the indexed terms. Thus an inverted index contains a substantial number of integers; as such integer compression can be used to compress the inverted index.

A number of studies present and assess compression schemes. In [7], Scholer et al. describe compression for fast query retrieval. Their empirical study showed that byte-aligned codes as opposed to bit-wise codes facilitate faster query evaluation. Another study [17] describes a technique for searching on compressed text. The technique is based on a compression scheme which uses a semi-static word-based model and a Huffman code where the coding alphabet is byte-oriented rather than bit-oriented. While the technique presented in this study uses fast and efficient compression schemes, the complexity of the schemes make it unfeasible to implement them in the time available for the project. Scholer et al. described compression of indices to have the following advantages:

- A compressed index requires less storage space.
- Compressed data makes better use of the available communication bandwidth. For fast decompression schemes, the total time cost of transferring compressed data and subsequently decompressing is potentially much less than the cost of transferring uncompressed data.
- Compression increases the likelihood that the part of the index required to evaluate a query is already cached in memory, thus entirely avoiding a disk access.

**Index maintenance**

The problem of fast index creation has received considerable research attention in the past few decades [18]. However, the problem of efficient index maintenance has had relatively little investigation. In a recent study [18], Lester, Zobel and Williams present and evaluate the three main strategies for index maintenance, focusing on addition of new documents.

The different index update strategies presented by Lester et al. are: in-place update, index re-merging and complete rebuild. With in-place update, the index has to be modified sequentially to include information about new documents. Typically, a document will contain hundreds of distinct terms, and hence in-place update will involve hundreds of disk accesses. This update strategy is thus inefficient if the index will be frequently updated which is the case for the indices in this project. The second strategy is to re-build the index from scratch when a new document arrives. This option is very expensive if it is done on a per-document basis, however if documents are cached, the costs may be acceptable. The last strategy involves dividing documents into blocks, constructing an index for each block and then re-merging. Experiments carried out by Lester et al. show that the latter index update strategy is the fastest approach for large numbers of updates. The open source search engines, Nutch and Swish-e, both use variations of the re-merge strategy [22, 23]. The index update strategy used in this project is also a variant of the re-merge approach.

### 3.2.3 Search engine cluster architectures

Clusters of low cost workstations are exploited by many large-scale Web search engines such as Google, Inktomi and FAST [20]. The architectures of these search engines require high performance, high scalability, high availability and fault tolerance. It is a challenging task to develop a cluster that meets these requirements. The difficulty is that most developments were done in competitive companies that do not publish technical details, thus very few papers discuss Web search engine architecture. A few implementations have explored cluster architecture design but with the aim of increasing query processing throughput.

**The Google Cluster Architecture**

The Google search engine architecture [4, 5 & 20] combines more than 15,000 commodity-class PCs with fault-tolerant software. Each of the PC has 256MB to 1GB of RAM, two 22GB or 40GB disks and run the Linux operating system. The nodes (PCs) are connected with 100Mbit Ethernet to a gigabit Ethernet backbone [5]. The architecture permits different queries to run on different processors. The index is partitioned into individual segments, thus queries are routed to the appropriate server based on which segment is likely to hold the answer.
Inktomi Architecture for Yahoo and MSN

The Inktomi search engine architecture serves many Web portals such as Yahoo, HotBot, Microsoft and others. It is a cluster based architecture utilizing Redundant Array of Independent Disks (RAID) arrays with special focus on high availability, scalability and cost-effectiveness. The large database (index) is distributed and queries are dynamically partitioned across multiple clusters. Each segment of the database handles a certain set of sub queries. Queries arrive at the manager where they are directed to selected workers. Each worker sends the queries to all workers that are tightly coupled with it through Myrinet [20].

AltaVista, Lycos and Excite Architecture

AltaVista, Lycos, and Excite make use of large Symmetric Multi-Processors (SMP) supercomputers. The use of large SMP allows fast access to a large memory space. The database is stored and processed on one machine. Processors handle queries independently on the shared database.

My Own Search Engine (MOSE)

Orlando, Perego and Silvestri [6] describe the design of their cluster-based search engine called My Own Search Engine (MOSE). Their aim is to increase query throughput by implementing an efficient parallelization strategy. MOSE uses a combination of a data and task parallel algorithm. The task parallel part is responsible for load balancing. It does so by scheduling the queries among a set of identical workers, each implementing a sequential Web search engine. The data parallel part partitions the database and allowing each query to be processed in parallel by several data parallel tasks, each accessing a distinct partition of the database. While the parallelisation strategy used by MOSE is powerful, and employed by successful search engines such as Google [4], it does not mention anything about keeping the indices fresh.

Yuntis

Lifantsev and Chiueh [9] describe Yuntis, a working search engine prototype. One of the goals of Yuntis is to utilize clusters of workstations to improve scalability. A Yuntis node runs one database worker process that is responsible for data management of all data assigned to that node. When needed, each node can also perform crawler tasks. Yuntis differs from this project in that the query nodes remain dedicated to responding to user queries. There is no dynamic allocation of nodes to the roles of querying and indexing. If the system is experiencing massive incoming data that needs to be indexed and there are no incoming queries, query nodes will be idle while the indexing nodes will be overloaded. In this case, the cluster will be under-utilized.

Existing search engines [3, 4, 6 & 9] employ static allocation of the query and index roles to nodes in a cluster. As pointed out above, this arrangement can lead to cluster under-utilization under certain system loads. This project investigates the use of dynamic allocation of the above roles. However, there are open questions, for example, will the overhead of dynamically changing work allocation outweigh performance gains obtained from the suggested cluster utilization strategy?
4. Design

This chapter describes the design of the indexing system. Firstly, the chapter gives an overview of the solution to the search engine project as a whole and indicates which parts the author was responsible for. Secondly, it describes what the system needed to include in order for the project to be deemed successful. Thirdly, it describes the techniques that were used to guide the implementation of the system. Finally, the chapter gives the algorithms employed by the system.

4.1 Overview of the Solution

The solution for the entire search engine system consists of two major subsystems, the indexing subsystem and the querying sub-system. The author of this report was responsible for developing the indexing module. The author’s project partner was responsible for the querying module. Documented in this chapter are the design details of the indexing system. The design of the querying system is beyond the scope of this report. The Figure below shows the search engine as a whole and the interfaces between the two sub-systems.

![Diagram of the search engine system](image)

**Figure 5: The design of the entire search engine system**

The highlighted parts of the diagram collectively make up the indexing subsystem; this collection will be referred to as the indexing system hereafter. The non-highlighted parts show the querying sub-system. The highlighted worker nodes show the nodes allocated to
indexing at a particular time. The nodes that are not highlighted are the nodes allocated to querying.

The interfaces through which the two subsystems are connected are in the form of files and a Load Balancer which is independently utilized by each subsystem. These interfaces are described below.

- The index – The query subsystem relies heavily on the index produced by the indexing module. The querying module needs to access the index before it can respond to user queries. The index is made available by the Indexing Dispatcher after which it can be accessed by the query subsystem.

- The id_urls.INFO file – The query module needs to respond to user queries with URLs as document ID are meaningless to users. This file contains the ID-URL mappings of all the documents that have been indexed by the system.

- The Load Balancer – The role of the load balancer is to monitor the load averages on the nodes allocated to indexing and querying. Based on the observed load averages the Load Balancer reallocates nodes to indexing and querying. The Load Balancer writes the number of machines allocated to indexing to a text file and writes the number of machines allocated to querying to another text file. The indexing and querying dispatchers read these files to determine the worker nodes that are allocated to indexing and querying respectively.

4.2 Features of the Indexing System Solution

This section describes the features of the indexing solution that were required for the indexing system to be deemed successful. It was already mentioned in Chapter 2 that search engines have demanding computing requirements, due to the large amounts of data that they have to deal with. For this reason, it is important that the system developed be able to run on a cluster of workstations. Furthermore, the system needs to be both correct and able to scale to large amounts of data.

4.2.1 Inverted indices

Inverted indices are the de facto standard for providing fast responses to queries over large documents. It is thus vital that the system employs inverted indices. The system should ensure that the indices produced are correct. Furthermore the program needs to be deterministic; it should produce the same output for the same input. This is an important requirement because if the program is not deterministic, it may produce incorrect indices and hence it will not serve any useful purpose to users. It is also important that the inverted indices are stored in a manner that ensures their fast retrieval. The developed system should avoid unnecessary expensive disk access.

4.2.2 Fresh indices

The main problem being investigated is that of outdated results. The system should have the ability to update indices after they have been created. It should be able to efficiently do so by utilizing optimal data structures.
It is vital that the index update algorithm be reasonably fast so that changes are reflected in the indices as soon as possible. A solution that wipes out the entire index at the beginning of each indexing cycle is not likely to perform well. This is because there is a huge number of documents that need to be indexed. Furthermore, this solution is likely to do a lot of wasteful re-indexing of documents that did not change since the last indexing cycle. Through incremental crawling, the system should keep track of newly created documents, and documents that have gone through changes since the last crawling cycle. The indexer should run as often as possible in order to ensure that indices reflect the current state of the indexed pages.

4.2.3 Load balancing and consistency
If the load is not properly distributed among the machines available for indexing, the performance of the system may degrade to that of a single machine. It is therefore a vital requirement that the system ensures that the load is reasonably balanced among the machines allocated to indexing at any time. This should be done based on the amount of data that needs to be indexed in order to ensure that none of machines are left idle while others are overloaded with work.

Moreover, the system needs to ensure that no two machines work on the same portion of data during one indexing cycle. In addition to ensuring that each document is uniquely identified within the system, this will also ensure that system resources are utilized efficiently and not wasted on work that has been done already or is currently being done on another machine. To further ensure consistency, the system should ensure that after updating the indices, documents remain uniquely identified in the system. Thus, each document should only be associated with a single document identifier even after updating the indices.

4.2.4 Scalability
All practical search engines index large amounts of data. It is vital that the system be able scale to at least hundreds of thousands of documents. Thus, the solution should avoid using slow algorithms that are likely to severely degrade system performance with increasing document numbers. Instead, data structures that are optimal for the problem at hand should be utilized. Furthermore, the system should avoid communicating very large messages among the machines. Too much communication can degrade performance with increased numbers of processors. Where communication cannot be avoided the system should employ measures to reduce the sizes of the messages that are sent.

4.2.5 Cluster utilization
One of the major project requirements of the project is the ability to efficiently utilize the cluster under different work loads. The system should thus monitor the work loads on all machines allocated to indexing to ensure that the cluster as a whole is used in the most efficient manner. The system should attain correct dynamic allocation of nodes to indexing such that there is minimal cluster under-utilization. It is also vital that the machines are allocated where they are needed most.
4.3 Overall Indexing System Design

This section describes the overall design of the indexing system. In order to make the system easy to debug and to make the project easily extensible, the system was divided into six main modules: the crawler, the parser, the stemmer, the actual indexer, the updater and the dispatcher. These components interact with one another to achieve the system functionality. The block diagram in Figure 6 below shows the overall structure of the indexing system and the flow of data through the system.

![Figure 6: The core components of the Indexing system](image)

To achieve parallel indexing the components in Figure 6 are distributed on a cluster. The Crawler and the Dispatcher components are executed by the cluster machine with the smallest rank; rank 0. The webpages are stored in the local disk of the machine with rank 0. The Indexer and Updater are executed by all the machines allocated to indexing at a particular time. All machines that run the Indexer and Updater create indices on their local disks which are merged by the Dispatcher to create the main index. Details of how this works are given later in the chapter. The Indexer and the Updater invoke both the Porter stemmer and the HTML parser.

From the diagram in Figure 6, it can be seen that the Dispatcher is the central component responsible for invoking the other components of the system. The Indexer and Updater components index the HTML documents that are made available by the crawler. For reasons already mentioned in Chapter 3, the system employs an existing crawler, GNU Wget. GNU Wget is a free software package for retrieving files using HTTP, HTTPS and FTP. It is a non-interactive command line tool [24].

The Indexer module creates an index from scratch whereas the Updater module updates an existing index based on newly available data after the last time indexing was performed. Both the Updater and Indexer modules use the HTML parser to extract HTML tags from documents before they are indexed. Extremely common words (stop-words) are excluded from indexing and all terms are case-folded to lower case. In addition, all terms are converted to canonical forms using the Porter stemming algorithm.
A C/C++ implementation of the stemming algorithm obtained from the Website of the author of the algorithm was used for this purpose.

4.3.1 The structure of the Index
As mentioned in Chapter 3, inverted indices are employed by nearly all practical search engines to facilitate fast response times to queries [16]. As a result, the indexer module that was developed in this project keeps inverted indices as a means of improving query response times. The inverted indices are stored in separate files, whereby each term has its own inverted file, with the actual term being the file name. Each inverted file contains postings about which documents it occurs in and the frequency of such occurrences. The structure of the postings in the inverted files is as follows:

```
docId  termFrequency
......
......
docId  termFrequency
```

The docId is a unique integer identifier for a document, and termFrequency is the number of occurrences of a term in the document with the specified document identifier. When the Indexer is first launched, inverted files are created completely from scratch. Subsequent calls to the Indexer module result in additions of new document information to the inverted files. The Updater is concerned with documents that have been modified or created since the last indexing cycle. For documents that changed, new information is added to the bottom of the appropriate inverted files. Obsolete postings are left unaltered. This is done in order to avoid lots of expensive disk access otherwise required to modify terms frequencies for individual terms. With this updating strategy, the latest postings are always at the bottom and obsolete ones always at the top. It would be desirable to have an update strategy with the opposite effect, where the latest postings are always at the top. However, currently no file system permits adding new content to the top of the file. A manual solution to obtain this effect is likely to use a lot of undesired disk accesses and hence result in an inefficient solution.

4.3.2 The ID-URL index
Every document encountered by the system for the first time is assigned a document identifier (ID), which is used to refer to that particular document. However, these IDs are meaningless to end users. Thus a query system needs to show Uniform Resource Locators (URLs) in the results returned to users. To address this, the indexing system keeps an ID-URL Index which is maintained by the Dispatcher. A new ID-URL index is created each time the Indexer module is called. In contrast, when the Updater is launched, the ID-URL index is updated accordingly, ensuring that each URL is associated with a single and unique ID.
The structure of the ID-URL index is as follows:

<table>
<thead>
<tr>
<th>docId</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>docId</td>
<td>URL</td>
</tr>
<tr>
<td>......</td>
<td></td>
</tr>
<tr>
<td>......</td>
<td></td>
</tr>
<tr>
<td>docId</td>
<td>URL</td>
</tr>
</tbody>
</table>

4.4 Algorithms

This section outlines the algorithms that were used for parallel indexing. Included are the master-slave algorithm, the indexing algorithm and the index update algorithm.

4.4.1 Master-slave approach and algorithm

A master-slave approach was used to achieve parallel indexing. The idea behind this approach is that one process is responsible for coordinating the work of others. This mechanism is particularly useful when there is little to no communication between the slave processes and when the amount of work that each slave has to perform is difficult to predict [14]. Both of the above cases apply to the task of indexing, as a result the model was chosen to accomplish parallel indexing. Furthermore, the master-slave approach fits well to tasks that are embarrassingly parallel. A computation task is embarrassingly parallel if it can be divided into a number of completely separate tasks, each of which can be executed by a single processor. In the case of indexing, slave processes do not need to communicate with one another, hence indexing can be classified as an embarrassingly parallel problem. Figure 7 shows how the Master-slave approach was applied to the indexing system.

As can be seen in the Figure, worker processors communicate with the master processor but there is no communication among individual workers. The master starts off with determining how much data needs to be indexed, it then divides the data among the slaves which then start indexing their respective portions. The slaves work independently on their data portions. After indexing is complete, all workers send their indices to the master processor where the indices are merged to form the final index.
In order to achieve parallelism, the computational work that needs to be done has to be partitioned among the available processors. This project employs a data parallel decomposition strategy. With data parallelism all processors perform the same task on different data as opposed to task parallelism where processors do different tasks. The data made available by the crawler is subdivided into data portions. Data portions are made up of an equal number of directories. This is not an ideal split as it does not ensure load balancing. This is because the number of files in different folders may vary considerably. Furthermore, the file sizes in different folders are not necessarily the same. However, considering the alternative solution of putting a burden on the master to do a per-file allocation of work, one opts for the former approach in order to avoid making the master processor a performance bottleneck. A data decomposition strategy that is guaranteed to achieve near ideal load balancing is one that allocates an equal number of files to each processor. The latter strategy was considered by implementing it for this project, however due to the solution’s slow execution speeds on large data sizes, it was scrapped in favour of the per-directory decomposition strategy.

The resulting master-slave algorithm is illustrated in Figure 8 below. The algorithm is divided into two parts, the part that is executed by the master and the part that is executed by the slaves. In the figure, the startID the master sends to the slave indicates the starting document ID for the indexing cycle. Initially this number is set to 1. The purpose of this startID is to ensure document ID uniqueness. Details of how the start ID is communicated among the slaves are given later in the section.

1. The Master
   - Send startID to slave with smallest rank
   - While there is still data to be indexed
     - Partition the data into chunks
     - Send chunks to indexer nodes
     - Receive indices from slaves
     - Merge the indices

2. The Slaves
   - While there are chunks available
     - Receive a chunk
     - Receive starting ID from slave with the rank one less than this slave's rank
     - Index the chunk
     - Send the resulting indices to master

Figure 8: An outline of the master-slave algorithm

At the beginning of the program, the master determines the approximate amount of data that is available for indexing by counting the number of directories available. These directories are then queued for indexing. A user specified parameter, the reallocation interval indicates how often reallocation of nodes to indexing and querying changes. The reallocation interval determines the minimum number of directories per chunk that each slave can work on.
The equations below illustrate how this works.

\[
\begin{align*}
\text{alldirs} & = \text{all directories available for indexing} & (1) \\
\text{dirs\_per\_proc} & = \text{alldirs}/\text{num\_slaves} & (2) \\
\text{rem\_dirs} & = \text{all\_dirs}\%\text{num\_slaves} & (3) \\
\text{chunk\_size} & = \text{dirs\_per\_proc}/\text{reallocation\_interval} & (4)
\end{align*}
\]

If the number of slaves is not a perfect divisor of the total number of directories, the remaining directories are allocated to the processor with the lowest rank, this is the number computed by equation 3. From equation 4 it is clear that the more often reallocation occurs, the smaller the chunk size.

Before a slave begins indexing, it receives a copy of the directories that it has been allocated. The directories are not sent one by one, but they are packed into a single directory and compressed using GNU zip (gzip). Compressing data before transmitting has advantages over sending huge amounts of uncompressed data. As Scholer et al. [7] put it, compressed data makes better use of the available communication bandwidth. For fast decompression schemes, the total time cost of transferring compressed data and subsequently decompressing is potentially much less than the cost of transferring uncompressed data.

Upon receipt of the data, a slave unpacks and decompresses the data. It then begins indexing and it will eventually send its results to the master where the results will be merged with those of other slaves. Merging of indices occurs by merging the contents of two directories and details of how this occurs are given below.

Although the master and the slaves in the cluster used in this project all mount a shared directory, local file systems for the master and slaves are used instead. Thus explicit message communication is required to send data between the master and the slaves. This was done in order to develop a flexible solution that can work even in the absence of a shared directory between the master and the slaves.

Both the master and the slave programs use auxiliary files and directories with predefined names during the program execution. Table 1 shows the files and directories that the master keeps during the program execution.
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Contents</th>
<th>Persistence</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>index</em></td>
<td>Directory</td>
<td>The root directory created by the master contains all sub-directories and files listed below.</td>
<td>Remains after program returns</td>
</tr>
<tr>
<td>_id_urls.INFO</td>
<td>File</td>
<td>Contains ID to URL mappings of all indexed documents.</td>
<td>Remains after program returns</td>
</tr>
<tr>
<td>index_all</td>
<td>Directory</td>
<td>Has inverted files of all documents that have been indexed.</td>
<td>Remains after program returns</td>
</tr>
<tr>
<td>last_ID</td>
<td>File</td>
<td>Has a single integer, to indicate the starting ID for the next indexing cycle. This is to ensure that ID is unique.</td>
<td>Remains after program returns</td>
</tr>
<tr>
<td>_indexfile_i, where i is the rank of a slave processor</td>
<td>Directory</td>
<td>Each slave sends this directory to the master. It contains inverted files of the documents indexed by the slave processor of rank i.</td>
<td>Deleted before program returns.</td>
</tr>
</tbody>
</table>

Table 1: Files and directories created by the master program

The contents of the index_all directory are created by merging the contents of index_all directory with all _indexfile_i directories that are sent to the master by the slaves. When indexing happens for the first time, the contents of the first _indexfile_i are simply copied into the index_all directory. For all subsequent _indexfile_i directories, merging occurs by copying the contents of the inverted file in the _indexfile_i to the end of the inverted file in the index_all directory with the same name, for all inverted files appearing in both the index_all directory and the _indexfile_i directory. Inverted files that only appear in the _indexfile_i directory are simply copied into the index_all directory.

The _id_urls.INFO file kept by the master is a result of merging the id_urls.INFO files produced by the slaves. The Table 2 shows files and directories created by each slave process during the program execution.
In order to ensure application-wide unique document IDs each slave processor determines the number of IDs it needs before it begins indexing. Each slave processor counts the number of HTML documents that are in the chunk to be indexed. The slave processor with the smallest rank receives the starting ID from the master processor. The starting ID from master is 1, if indexing is happening for the first time or the last ID number from the previous indexing cycle incremented by 1. Having counted the number of files contained in the chunk to be indexed, the slave processor with the smallest rank then adds the number of IDs it requires to the starting ID. The result incremented by one is the starting ID for the processor with the next smallest rank and this number is sent to the appropriate processor via an MPI message. This operation is carried out until all slave processors have a starting ID. Details of how ID uniqueness is maintained during index updates are described in the index update algorithm sub-section. Figure 9 is an illustration of how ID uniqueness is ensured.

Table 2: Auxiliary files and directories created by slave processors

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Contents</th>
<th>Persistence</th>
</tr>
</thead>
<tbody>
<tr>
<td>_store_i, where i is the rank of the slave</td>
<td>Directory</td>
<td>The root directory created by each slave contains all sub-directories and files created listed below.</td>
<td>Deleted before program returns.</td>
</tr>
<tr>
<td>_id_urls.INFO</td>
<td>File</td>
<td>Contains ID to URL mappings of all documents indexed by a slave. This is sent to the master for each indexed chunk.</td>
<td>Deleted before program returns.</td>
</tr>
<tr>
<td>_indexfile_i, where i is the rank of a slave processor</td>
<td>Directory</td>
<td>It contains inverted files of the documents indexed by the slave processor of rank i. This is sent to the master for each indexed chunk.</td>
<td>Deleted before program returns.</td>
</tr>
<tr>
<td>last_ID</td>
<td>File</td>
<td>Created by the slave process with the largest rank, it contains the starting ID for the next indexing cycle. This is file is sent to the master.</td>
<td>Deleted before program returns.</td>
</tr>
</tbody>
</table>
4.4.2 The indexing algorithm

This section describes the indexing algorithm that was employed to achieve indexing at the basic level. The algorithm counts the number of occurrences of terms in each document. Each term is converted to its canonical form using Porter’s stemming algorithm. An implementation of Porter’s Algorithm obtained from the Website of the author of the algorithm was used in this project. The indexing algorithm also ignores all extremely common words and converts all words to lower case. The data structure used to achieve indexing is the hashtable. A hashtable has the advantage that it provides constant time lookup on average, regardless of the number of entries in the table. Figure 10 outlines the indexing algorithm used.

- For each HTML file
  - Increment docId counter
  - Read in the file
  - Store (docId, URL) in the ID-URL mapping file
  - Remove all tags, punctuations and spaces
  - Fold case and remove stop words
  - Apply stemming to each word
  - Count number of occurrences of each word (TF)
  - Add (docId, TF ) to a hashtable keyed on words

- Save the Hashtable to disk.

4.4.3 The index update algorithm

In order to ensure that the index produced is up-to-date, the indexing system needs to update changed pages as pages can be deleted, new data can be added to pages, and new pages can be created. Index updating proved not to be a trivial problem. The solution developed to achieve index updating looks at the date a document was last modified to
determine whether the document needs to be re-indexed or not. This solution is far from optimal however it is an improvement over the alternative solution of wiping out the entire index at each indexing cycle and re-indexing from scratch. Furthermore, this solution does not take into account page deletions, but it does cater for addition of data to pages and creation of new pages.

The index update algorithm ensures that even after updating, each document is only associated with a single unique document identifier. This is achieved by communicating the relevant data between the master process and the slave processors. The main issue is that the slave processes cannot determine pages that have changed since the last indexing cycle by looking at the date of last modification. This is because all files sent to the slave processes will reflect the date of last modification as the date the file arrived in slaves’ local file systems. This has the undesired effect of re-indexing all files, even those that did not change since the last indexing cycle. Below is a description of the parallel index update algorithm used in this project.

![Figure 11: An outline of the index update algorithm](image)

Like the initial parallel indexing algorithm described in Section 4.4.1, the master starts off by determining the number of directories available for indexing. Since the master is responsible for running the crawler, it can easily determine which pages changed since
the last crawling phase. Thus, before the master sends a chunk to a slave, it sends the IDs and URLs of all documents that changed since the last modification dates, including all newly added documents. Upon receipt of the ID – URL file of modified files, a slave saves this information into a hashtable keyed on URLs. When a slave is indexing a document, it first checks to see if the document has been indexed before, in which case it will have been allocated an ID. If a file has not been indexed before, its ID entry in the ID-URL hashtable will be negative; in this case the slave will assign a new unused ID to the document. The document is indexed using the same algorithm as in Section 4.4.2 and the indices are sent to the master where they are merged.

4.5 The Parser

A simple HTML parser was developed in order to extract the content of the HTML files downloaded by the crawler. The parser uses no special algorithms and does not utilize additional information of HTML documents. The main operation done by the parser is to ignore all HTML tags and to store the data enclosed within the tags. It also ignores all script code snippets embedded within HTML documents.

4.6 Package Diagrams

This section contains the high level package diagram for the indexing system. Figure 12 shows the package diagram. A description of each package is given in Table 3.

![Package diagram for the indexing module](image-url)
<table>
<thead>
<tr>
<th>Package name</th>
<th>Description</th>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utils</td>
<td>Contains auxiliary classes needed by the indexer and the updater.</td>
<td>fileops.cpp hashtable.cpp porter.cpp</td>
</tr>
<tr>
<td>Updater</td>
<td>Contains classes for index updating.</td>
<td>update.cpp</td>
</tr>
<tr>
<td>Indexer</td>
<td>Contains classes for index creation.</td>
<td>indexer.cpp create.cpp</td>
</tr>
<tr>
<td>Parser</td>
<td>The HTML parser.</td>
<td>parser.cpp</td>
</tr>
<tr>
<td>Dispatcher</td>
<td>The dispatcher.</td>
<td>dispatcher.cpp loads.cpp</td>
</tr>
</tbody>
</table>

Table 3: Description of the packages that make up the system

4.7 Class Diagrams

Figure 13 shows the detailed class diagram for the system. A short description of each of the classes is given in Table 4 below.

<table>
<thead>
<tr>
<th>Class</th>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileops.cpp</td>
<td>Utils</td>
<td>This class contains methods that deal with file operations, i.e. copy a file from one to another, sending a file, etc.</td>
</tr>
<tr>
<td>hashtable.cpp</td>
<td>Utils</td>
<td>Represents a hashtable data structure, this class has been tuned to work for indexing.</td>
</tr>
<tr>
<td>porter.cpp</td>
<td>Utils</td>
<td>An implementation of Porter’s stemming algorithm obtained from [26].</td>
</tr>
<tr>
<td>update.cpp</td>
<td>Updater</td>
<td>Class for updating an inverted index.</td>
</tr>
<tr>
<td>indexer.cpp</td>
<td>Indexer</td>
<td>Implements an indexing algorithm.</td>
</tr>
<tr>
<td>create.cpp</td>
<td>Indexer.cpp</td>
<td>Class for creating an inverted index for a set of HTML documents.</td>
</tr>
<tr>
<td>parser.cpp</td>
<td>Parser</td>
<td>Parses HTML documents.</td>
</tr>
<tr>
<td>dispatcher.cpp</td>
<td>Dispatcher</td>
<td>Class for coordinating the work of slave processes.</td>
</tr>
<tr>
<td>loads.cpp</td>
<td>Dispatcher</td>
<td>Class for monitoring the load averages of indexing and querying machines and for allocating cluster nodes according to the load averages.</td>
</tr>
</tbody>
</table>

Table 4: Description of classes belonging to the Indexing system
Figure 13: Class diagram for the indexing system
5. Experimental Design

After the indexing system was fully developed, it was necessary to assess the system in terms of performance and cluster utilization. This chapter details the experiments which were conducted in order to evaluate the system. The chapter first explores the basis for the formulated hypothesis and then proceeds to state the experimental and null hypotheses. Criteria for acceptance or rejection of the hypothesis are then discussed before detailing the experiments that were carried out.

5.1 Basis of the Hypothesis

On a relatively small cluster, where they are about ten nodes, dividing the cluster nodes between two tasks can be a difficult job. By dividing the already few nodes among two tasks, it is guaranteed that each task will only have few nodes allocated to it. However, for the tasks in question in this project, the tasks of querying and indexing, there is a useful property that can be taken advantage of, namely that the amount of system resources required by indexing and querying unpredictably change over time. Indexing happens at discrete intervals, and thus there are peak times when indexing requires maximum system resources and idle times when there is no data to be indexed. On the other hand, the system resources required by querying are based on the number of queries the system receives. Thus for querying, peak times occur when users pose a lot of questions to the system and low-traffic times occur when there is a small number of questions to respond to. It is therefore clear that the ideal split of nodes among the tasks of indexing and querying changes over time. If the ideal split is attained at every stage, both indexing and querying would achieve better performance. This assumption forms the basis and reasoning behind the hypothesis described below. If the assumption that the ideal split of nodes between the task of indexing and queries changes over time is in fact true, then the hypothesis posed below has a high probability of being true.

5.2 The Hypothesis

The aim of this section is to clearly state the hypothesis as well as the associated null hypothesis. The hypothesis stated in this section will guide the interpretation of results and provide a basis for conclusions. The hypothesis that the experiment sought to support and its associated null hypothesis can be stated as follows:

Experimental Hypothesis: \( H_1: \) “Dynamic allocation of nodes to indexing results in faster indexing and better system resource utilization than static allocation”

Null hypothesis: \( H_{10}: \) “There is no difference between dynamic and static allocation of nodes to indexing, in terms of indexing times and system resource utilization.”

For the null hypothesis to be rejected in favour of the experimental hypothesis, the experiment carried out needed to yield results showing that system resources are better utilized to achieve fast indexing with dynamic allocation than with static allocation. However, if the results indicated that indexing times are the same or shorter with static
allocation than with dynamic allocation, then the null hypothesis would be accepted. This is because there would be insufficient evidence to support the experimental hypothesis.

5.3 The Experiments

This section describes the experiments that were carried out to test the hypothesis stated above. The metric of interest in the experiments was the time taken to perform indexing on a set of documents. This was measured using C++ timing functions.

The core part being tested is dynamic allocation versus static allocation of nodes to indexing and querying. The experiment was set up in order not to bias the results in favour of any of the two parts. The dispatcher part of the indexing system was the part responsible for dynamic allocation of nodes to indexing as described in the design chapter. When data is available the dispatcher divides it equally among the nodes available for indexing. The division is done based on the number of folders for reasons documented in the design chapter. The dispatcher takes in three user-specified parameters; the last two were used for testing the hypothesis. The first parameter indicates whether to run the indexer program or the updater program. The second parameter indicates the starting number of machines allocated to indexing, the third parameter indicates the number of times the system will check the load averages on the cluster machines allocated to indexing and that of the nodes allocated to querying and change the node allocation if it is necessary. In other words, the third parameter indicates the reallocation interval. With a greater reallocation interval, the dispatcher monitors the load more often and thus it is more likely to get a more accurate split of the machines.

The UNIX virtual file /proc/loadavg was used to obtain the load averages on individual machines. The /proc/loadavg file contains 6 numbers only the first three were of interest in these experiments. These first three numbers are load average figures giving the number of jobs in the run queue or waiting for disk I/O averaged over 1, 5 and 15 minutes. Due to the fast execution speed of the program, the load average averaged over 1 minute was used in the experiment.

A load monitoring program was executed in parallel with the indexing program to allow every machine on the cluster to read the contents of its /proc/loadavg file and to send these contents to the dispatcher every 80 seconds. The 80 second interval was chosen by trial and error. The load monitor reallocates the nodes based on the load average on the indexing and querying machines. To determine which machines are allocated to querying, the load monitor makes use two files, one which indicates how many nodes are allocated to querying and one that indicates how many nodes are allocated to indexing. These files are initially created with the number of machines set to 2.

Dynamic allocation of nodes is solely based on the ratio of the load average of the machines allocated to indexing over the load average of the machines allocated to querying. If the ratio obtained is greater than 1.5, then there is an additional machine allocated to indexing. The number of machines is only increased by one due to the small size of the cluster and to smooth the effect of changing loads.
With the load monitoring program running in the background, the experiment proceeded to measure the metric of interest under different system loads. The effect of dynamic allocation on the time taken to index data was measured with different data sizes. With the aid of graphs, these times were compared to the times obtained with static allocation of nodes to indexing. To attain static allocation, the dispatcher was executed with the third parameter set to 0. The effect of how often the dispatcher monitors the load on the time taken to index was also tested. This was done by varying the reallocation interval with different data sizes. In order to check how system resources are utilized, the load averages over all the cluster machines were calculated both with dynamic and static allocation.

To monitor the freshness of the index, the index update program was executed with different amounts of updated documents. The metric of interest was the time taken to update the inverted index. The size of the inverted index just before updating started was also taken into account as an important variable. The results were analyzed with the aid of graphs which are shown in the results chapter.

5.4 Equipment

In order to allow for correct replication of the experiment, the computer equipment used during the experiment is described below.

The cluster on which all experiments were conducted is as follows:

- 13 Machines in total
- Operating system: Gentoo Linux (kernel version: 2.6.17 on machines 1 to 3, and 2.6.15 on machines 4 to 13)
- CPU: 3GHz Intel Pentium 4 with 1MB cache
- Main memory: 512 MB
- Disk capacity: 80GB on each machine
- The cluster is behind a firewall with disk capacity of 145GB
- Network Interconnect: Gigabit Ethernet (1000 megabits per second)

The cluster was accessed remotely by using a Secure Shell (SSH) client called PuTTY\(^1\). The Windows machine on which Putty was run is running the Windows XP professional operating system. At the time the experiments were conducted, the MPI implementation on the cluster is LAM MPI version 7.0.6

\(^1\) PuTTY is a free implementation of Telnet and SSH for Win32 and Unix platforms, along with an xterm terminal emulator. [http://www.putty.nl/](http://www.putty.nl/)
6. Results

This chapter describes the results of the experiments that were conducted for the indexing sub-system. These results are not absolutely rigorous but provide indications of some degree of success and serve as a starting point for future, more comprehensive evaluation.

6.1 Index Creating Program

This section presents the experimental results obtained from the experiments that were carried out on the indexing creating program. It is important to mention that due to the fact that the querying program was not available at the time the experiments were conducted, all tests assumed a constant querying load average of 0.2. The results for the index updating program are presented in Section 6.2.

6.1.1 Reallocation interval

It has been previously mentioned that one of the parameters of the dispatcher program is a number that indicates how often the dispatcher monitors the load on the machines allocated to indexing and querying. The reallocation interval has a significant impact on how well the dispatcher attains a good split between the indexing and querying machines. The explanation for this is that the more often the dispatcher checks the load averages on the machines, the more it is likely to obtain a true picture of the amount of work the machines are doing. An experiment was conducted to determine the actual effect on the time taken to index data. Figure 14 to 16 show the results of the experiment performed on different data sizes. The data used to plot all graphs can be found in the appendix. In all cases the starting number of machines allocated to indexing is 3, i.e. if there is no reallocation then indexing will remain allocated to 3 machines.

![Graph showing the effect of reallocation interval on the time to index 119 423 files](image)

Figure 14: Effect of reallocation interval on the time to index 119 423 files
Figure 14 evidently shows that the reallocation interval has a great impact on the time taken to index data. A reallocation interval of 0 is equivalent to static allocation of machines to the roles of querying and indexing. It can be seen in the figure that indexing takes longer with static allocation than it does with dynamic allocation. The reason for this is that as the load average on the indexing machines increases, the number of machines allocated to indexing also increase. Indexing took the least amount of time at the reallocation interval of 7. At this point the optimal number of machines for indexing is reached. After the optimal number of machines has been realized, additional machines do not provide any benefit. This behaviour can be seen in the figure where the time to index starts to increase with increasing reallocation interval. The reason for this behaviour is that after the optimal reallocation interval, the increase in communication time outweighs the decrease in execution time brought about by the increase in the number of machines.

The optimal reallocation interval varies from data size to data size. Figures 14 and 16 illustrate this. For example, in Figure 14 with just over 100 000 files, the optimal reallocation interval is 7, whereas in Figures 15 with about 50 000 files and 16 with about 10 000 files, it is 2 and 4 respectively.

![Graph](image)

**Figure 15: Effect of reallocation interval on the time to index 50149 files**
In Figure 16, it can be seen that the optimal reallocation interval is reached at 2. Thereafter, increase in the reallocation interval does not provide performance benefits.

### 6.2.2 Dynamic versus static allocation

Figure 17 below shows performance of static and dynamic allocation. Dynamic allocation was performed multiple times with different reallocation intervals.

![Figure 17: Performance of static and dynamic allocation](image)
The starting number of machines allocated to indexing used in all dynamic allocations scenarios in Figure 17 is 3. Similarly, the number of machines used for static allocation is 3. This is to illustrate that if it is assumed that indexing does not happen as often as querying, most of the machines will be allocated to querying and the remaining few will be allocated to indexing. Although it is in fact correct to say that indexing occurs less frequently than querying, there are cases when large amounts of data need to be indexed and indexing becomes more computationally intensive than querying.

From Figure 17, it can be seen that for small data sizes, the time taken to index data for dynamic and static allocations is almost the same. However, as the size of the data increases, the static allocation line is significantly above the dynamic allocation lines. Furthermore, the Figure shows that with the right reallocation interval dynamic allocation can realize much faster execution speeds than static allocation.

6.2.3 System Resource Utilization

Figure 18 shows how the number of machines allocated to indexing as the ratio of the load average on the indexing machines over the load average of querying machines changes.

![Graph: Ratio of Indexing/Querying versus Number of Indexing nodes](image)

From Figure 18, it can be seen that for dynamic allocation, as the ratio increases more machines are allocated to indexing. Dynamic allocation allows machines to be allocated where they are most needed. With static allocation the number of machines allocated to indexing remains constant. Static allocation can lead to some idle machines in the system while leaving some of the machines overloaded.
To further illustrate that dynamic allocation achieves better resource utilization than static allocation, a graph that indicates the number of idle machines as a function of indexing over querying load average ratio. The graph was drawn from the data used to plot Figure 18. The number of idle machines was calculated with the information that the load average on the querying machines is constant at 0.2, with only 2 querying machines doing work and the rest are idle.

![Graph showing number of idle machines versus indexing/querying load average ratio.](image)

Figure 19: Number of idle machines with dynamic and static allocation

It is evident from Figure 19 that as dynamic allocation progresses, the number of idle machines in the cluster goes down. Whereas with static allocation, the number of idle machines remains constant and that can lead to cluster under-utilization.

### 6.2 Index Updating Program

Figure 20 shows performance of the index updating. Dynamic allocation was performed with the optimal reallocation interval for the respective data sizes. The optimal reallocation intervals were obtained in Section 6.1.
From Figure 20, it can be seen that index updating is faster with dynamic allocation than with static allocation even when the size of the inverted index being updated is big. The index size of 128 Megabytes contains 5148 inverted files obtained by indexing of 100,000 files.

6.3 Discussion

This section analyses the results obtained from the experiments. Included is the support for the experimental hypothesis and the scope of the results.

6.3.1 Index creating

The results obtained from the experiments conducted on the indexing system provide enough support for the experimental hypothesis (H1). This means that dynamic allocation results in faster indexing and better utilization of system resources. It is important to state the difference between just increasing the number of machines in the cluster and using dynamic allocation. Dynamic allocation monitors the cluster work load in order to allocate machines to the job that needs the most computational power at a particular time. On the other hand, increasing the size of the cluster uses static allocation of machines to the roles of indexing and querying. Thus the latter approach relies on assumptions about the computational power requirements of querying and indexing the cluster. Furthermore, increasing the cluster size is costly as it requires more machines to be purchased. Thus dynamic allocation is a more flexible and cost-effective approach than increasing the cluster size.
6.3.2 Index updating
The results shown in Section 6.2 provide evidence that index updating is also faster with dynamic allocation than it is with static allocation. This further supports the hypothesis stated in Chapter 5.

6.3.3 Scope and Limitations of Results
The experiments conducted in this project are relatively small scale in that the data sizes used were small compared to large amounts of data that practical search engines have to deal with. Thus in order for these results to be useful they need to be limited to the data sizes that were catered for in this project. Furthermore, the webpages used were only obtained from the uct.ac.za domain. It is possible that these pages are larger or smaller than average webpages. Therefore, experiments need to be duplicated with a larger and more varied data before results and outcomes can be generalized. As it has been mentioned, the experiments were conducted assuming a constant load average on the querying machines, thus it is vital that the experiments are conducted with varying load averages on the machines allocated to querying.
7. Conclusions

The aim of this project was to build a search engine with the following properties:

- Cluster-based.
- Capable of dynamically allocating the roles of indexing and querying according to the system load.
- Able to utilize system resources efficiently under different use-case scenarios.
- Scalable.

The problem which this project aimed to solve was clearly defined. It was established that solutions used by current search engines to handle the problem of outdated results are designed for large clusters. From background research, it became apparent that current search engines employ static allocation of nodes to the roles of querying and indexing. Having identified this problem, the design details of a system that uses a flexible solution that dynamically allocates cluster nodes to querying and indexing was presented. The software product of the design was realized by implementing a system that efficiently utilizes a cluster of workstations and allocates resources where they are most needed. The Indexing sub-system developed allows creating and updating an inverted index. The Indexing sub-system also permits specification of a reallocation interval which when optimally chosen can significantly reduce indexing time. Although the Indexing sub-system employed a parser that is only able to parse HTML documents, it can be extended as a subject of future work.

Experiments were conducted to test performance of the concept of dynamic allocation as part of the implemented system. Experimental results proved that dynamic allocation results in faster indexing and better utilization of system resources.

The project has therefore achieved the goals set out earlier by means of implementing a system that employs a flexible solution that dynamically allocates cluster nodes to the roles of indexing and querying.
8. Future Work

A complete indexing engine would not have been feasible to implement in the time allotted to the project. Thus there are a number of indexing optimizations that can still be added to the system. Although the project achieved its aims it was conducted on a small scale and thus yielded results that cannot be easily generalized to many situations. Future research is therefore required to improve on the solution presented in this project and to experiment with more and larger data sets.

8.1 Fault Tolerance

The system currently does not cater for network failures or failure of certain machines in the cluster. The system needs to continue working in the face of network failures. A possible improvement is to ensure that messages are received by their recipients by means of sending acknowledgements for every message sent or received. The sending processor will need to keep sending the message for a set maximum number of tries until the message is received. If a processor does not respond after the maximum number of sends, the system can assume that the machine is down and necessary reconfiguration can take place.

8.2 Index Compression

This project aimed to implement differential encoding or d-gaps as a means of index compression. However, due to the way index updating was carried out it was not feasible to implement differential encoding. The reason for this is that, the index update algorithm works such that the latest information is always at the bottom of the inverted file and thus duplicate ID could appear. With multiple entries appearing in inverted files, it is necessary to sort entries very time new information is added. This has a potential to dramatically increase the time to index.

8.3 A General-Purpose Parser

Due to time constraints, the parser implemented is very simple and only parses HTML documents. The parser can be extended to be general-purpose in that it can parse other types of documents such as XML, PDF, etc. Furthermore, the parser can be extended to handle metadata information in HTML files in order to improve the quality of the results that the indexing system can provide.

8.4 Load-Balancing

The system currently performs work allocation by dividing the number of available directories among the indexing machines. As mentioned already, an allocation based on the number of files was considered too slow. Thus a load-balancing algorithm that ensures proper balancing while not compromising execution speed is required. A possible solution is to enable the slaves in the master-slave algorithm to report to the master each time they are done working on a data chunk. This way the time spent idle by processors can be reduced as new data chunks are made available to processors shortly after they finish working on one data chunk.
8.5 Document Summaries

Currently the system only stores document IDs and URLs with this scheme only the URLs will be returned in the results to user queries. It will be useful to store document titles and summaries. Users can make use of document summaries to judge which results are most relevant to their contexts.

8.6 Index Update Algorithm

The algorithm used to update the inverted index is based on examining the date of last modification for every document. This is a naïve algorithm which can be improved on. A possible improvement on this is to develop an incremental crawler which is capable of clearly separating changed pages from those that did not change.
9. References


Appendix

The tables in this section show the data that was used to plot the graphs in the results chapter.

<table>
<thead>
<tr>
<th>Reallocation Interval</th>
<th>Time to index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>858</td>
</tr>
<tr>
<td>1</td>
<td>490</td>
</tr>
<tr>
<td>2</td>
<td>390</td>
</tr>
<tr>
<td>3</td>
<td>351</td>
</tr>
<tr>
<td>4</td>
<td>319</td>
</tr>
<tr>
<td>5</td>
<td>214</td>
</tr>
<tr>
<td>6</td>
<td>213</td>
</tr>
<tr>
<td>7</td>
<td>200</td>
</tr>
<tr>
<td>8</td>
<td>241</td>
</tr>
<tr>
<td>9</td>
<td>258</td>
</tr>
<tr>
<td>10</td>
<td>256</td>
</tr>
<tr>
<td>11</td>
<td>260</td>
</tr>
<tr>
<td>12</td>
<td>305</td>
</tr>
<tr>
<td>13</td>
<td>327</td>
</tr>
<tr>
<td>14</td>
<td>369</td>
</tr>
</tbody>
</table>

Table 5: Data used to draw graph in Figure 14

<table>
<thead>
<tr>
<th>Reallocation Interval</th>
<th>Time to index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>310</td>
</tr>
<tr>
<td>1</td>
<td>189</td>
</tr>
<tr>
<td>2</td>
<td>172</td>
</tr>
<tr>
<td>3</td>
<td>137</td>
</tr>
<tr>
<td>4</td>
<td>135</td>
</tr>
<tr>
<td>5</td>
<td>140</td>
</tr>
<tr>
<td>6</td>
<td>149</td>
</tr>
<tr>
<td>7</td>
<td>153</td>
</tr>
<tr>
<td>8</td>
<td>158</td>
</tr>
<tr>
<td>9</td>
<td>161</td>
</tr>
<tr>
<td>10</td>
<td>165</td>
</tr>
<tr>
<td>11</td>
<td>170</td>
</tr>
<tr>
<td>12</td>
<td>171</td>
</tr>
<tr>
<td>13</td>
<td>180</td>
</tr>
<tr>
<td>14</td>
<td>186</td>
</tr>
</tbody>
</table>

Table 6: Data used to draw graph in Figure 15
<table>
<thead>
<tr>
<th>Reallocation Interval</th>
<th>Time to index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>51</td>
</tr>
<tr>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>34</td>
</tr>
<tr>
<td>4</td>
<td>38</td>
</tr>
<tr>
<td>5</td>
<td>39</td>
</tr>
<tr>
<td>6</td>
<td>41</td>
</tr>
<tr>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>8</td>
<td>46</td>
</tr>
<tr>
<td>9</td>
<td>46</td>
</tr>
<tr>
<td>10</td>
<td>47</td>
</tr>
<tr>
<td>11</td>
<td>46</td>
</tr>
<tr>
<td>12</td>
<td>48</td>
</tr>
<tr>
<td>13</td>
<td>49</td>
</tr>
<tr>
<td>14</td>
<td>48</td>
</tr>
</tbody>
</table>

Table 7: Data used to draw graph in Figure 16

<table>
<thead>
<tr>
<th>Data size (Number of files)</th>
<th>Static</th>
<th>6 relocs</th>
<th>6 relocs</th>
<th>9 relocs</th>
<th>12 relocs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1658</td>
<td>8</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11901</td>
<td>24</td>
<td>30</td>
<td>37</td>
<td>38</td>
<td>43</td>
</tr>
<tr>
<td>35000</td>
<td>127</td>
<td>60</td>
<td>54</td>
<td>79</td>
<td>87</td>
</tr>
<tr>
<td>50149</td>
<td>310</td>
<td>140</td>
<td>113</td>
<td>130</td>
<td>148</td>
</tr>
<tr>
<td>119423</td>
<td>858</td>
<td>390</td>
<td>214</td>
<td>241</td>
<td>260</td>
</tr>
</tbody>
</table>

Table 8: Data used to draw graph in Figure 17

<table>
<thead>
<tr>
<th>Indexing/Querying</th>
<th>Dynamic alloc</th>
<th>Static alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.67</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3.5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5.05</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>6.29</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7.19</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>8.75</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 9: Data used to draw graph in Figure 18

<table>
<thead>
<tr>
<th>Data size (Number of files)</th>
<th>Static (index size =0)</th>
<th>Dynamic (index size =0)</th>
<th>Dynamic (index size = 128 M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1658</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>11901</td>
<td>28</td>
<td>27</td>
<td>29</td>
</tr>
<tr>
<td>35000</td>
<td>127</td>
<td>62</td>
<td>75</td>
</tr>
<tr>
<td>50149</td>
<td>310</td>
<td>124</td>
<td>129</td>
</tr>
<tr>
<td>119423</td>
<td>858</td>
<td>279</td>
<td>296</td>
</tr>
</tbody>
</table>

Table 10: Data used to draw graph in Figure 20