An Approach to Better System Resource Utilization for Search Engine Clusters

Calvin Pedzai
University of Cape Town
Rondebosch
Cape Town, South Africa
cpedzai@cs.uct.ac.za

Ndapandula Nakashole
University of Cape Town
Rondebosch
Cape Town, South Africa
nnakasho@cs.uct.ac.za

Hussein Suleman
University of Cape Town
Rondebosch
Cape Town, South Africa
hussein@cs.uct.ac.za

ABSTRACT
Better system resource utilization for search engine clusters can result in significant benefits. By allocating cluster machines to the job that requires the most computational power, indexing and querying both realize performance gains. In this paper we discuss an approach to better system resource utilization which was tested by implementing it in a cluster-based search engine. We test the approach on 100 000 webpages from the uct.ac.za domain. Our results show the benefits of enhanced system resource utilization in a search engine cluster.

Categories and Subject Descriptors
C.4 **[Performance of Systems]:** H.3.4 **[Information Storage and Retrieval]:** Systems and Software H.3.5 **[Information Storage and Retrieval]:** On-line Information Services

General Terms
Algorithms, Design, Performance

Keywords
Indexing, querying, search engine cluster, index updating, outdated results

1. INTRODUCTION
A search engine is a document retrieval system designed to facilitate navigation of large information environments such as the Web. A search engine is typically made up of three major components: a crawler, an indexing module and a query module. The crawler is responsible for downloading Webpages that are stored in a local store. The role of the indexing module is to record which words appear in each page to create an inverted index. The query module accepts search queries from users and performs searches on the indices.

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 [3].

Although there are a number of successful search engines, there exists problems that need to be addressed further. One such problem is that of returning outdated results. 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 [1]. 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, for example news websites.

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 [2]. However, this solution is not flexible as it cannot be effectively deployed on a small cluster consisting of about 10 machines.

In this paper we present a flexible solution that is deployable on both large and small clusters. The solution is based on dynamic allocation of indexing and querying roles to cluster nodes in order to optimize cluster utilization.

The rest of this paper is structured as follows: Section 2 details the anatomy of our search engine; Section 3 describes the experiments carried out on the search engine and the results that that were obtained; Section 4 reviews related works on the topic of search engine clusters; Section 5 has some concluding remarks. Finally, possible future work is proposed in Section 6.

2. ANATOMY OF OUR SEARCH ENGINE
2.1 System Overview
The search engine that was developed to test the approach in question was developed as two independent subsystems, namely the Indexing subsystem and the Querying subsystem. The search engine dynamically allocates cluster nodes to the roles of indexing and querying based on the system load. The allocation changes over time as the work load on the querying and indexing machines change. Figure 1 shows the architecture of the developed search engine. The highlighted parts of the diagram collectively make up the Indexing subsystem; the non-highlighted parts show the Querying subsystem. The highlighted worker nodes show the nodes allocated to indexing at a particular time.

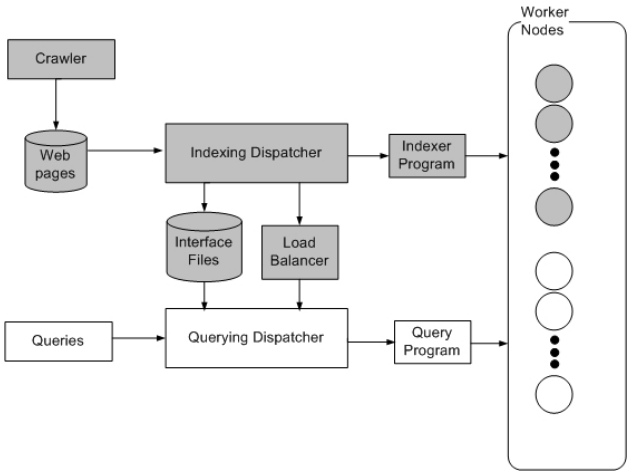

Figure 1: The architecture of the developed search engine

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 subsystem as the former needs to access the index before it can respond to 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. This file contains the ID-to-URL mappings of all the documents that have been indexed by the system. Identifiers (IDs) are needed by the indexing system as an efficient way to uniquely identify each indexed page.
- The Load Balancer– The role of the load balancer it 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.

2.2 The Indexing Subsystem

In order to make the system easy to debug and easily extensible, the indexing subsystem was divided into six main components, namely: 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 diagram in Figure 2 shows the overall structure of the indexing subsystem.

To achieve parallel indexing, the components in Figure 2 are distributed on a cluster. The Crawler and the Dispatcher components are executed by the cluster machine with the smallest rank which is 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

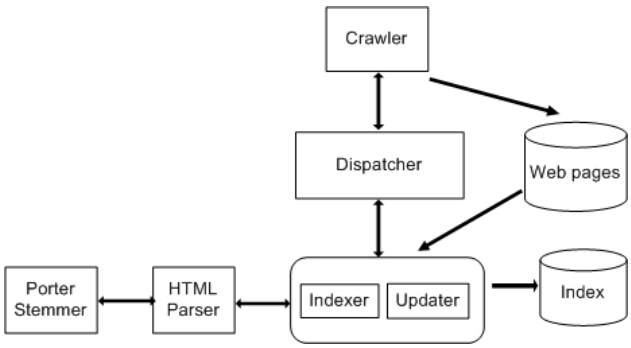

Figure 2: The core components of the Indexing subsystem

From the diagram in Figure 2, 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. The system employs an existing crawler, GNU Wget. GNU Wget is non-interactive command line tool for retrieving files using HTTP, HTTPS and FTP [4].

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 [5]. A C/C++ implementation of the stemming algorithm obtained from the Website of the author of the algorithm was used for this purpose.

2.3 The Querying Subsystem

The querying subsystem receives queries from users as a string of keywords highlighting the key aspects of the information that a user is looking for. These queries are fed through the users interface to the dispatcher for processing. Once they reach the dispatcher, the dispatcher has to decide which machine in the cluster will handle the query. This is done by simply reading from a file the number of machines in the cluster that have been allocated to querying by the load balancer.

The load balancer continually runs in the background constantly computing the load averages of the indexing and querying subsystems and deciding which nodes do indexing and which nodes do querying.

Once a cluster machine is chosen the query is sent off to the machine and the necessary index files are copied over. Each query is stemmed and stopped to improve on recall and precision. Once the query is sent to the worker node term weights for each document from the index files are used to compute the similarity of the document to the request. Once the computation and results

are done, a ranked list of documents is sent back to the Dispatcher to return to the user.

To generate a large number of queries for testing, an external program was used which simulates real world queries based on the Web pages which have been indexed. It randomizes the length of the query and the keyword selected.

3. EXPERIMENTAL RESULTS

3.1 Equipment

We conducted experiments on a cluster of 13 Gentoo Linux PCs interconnected by a Gigabit Ethernet network. Each PC is equipped with a 3GHz Pentium 4 processor, 512 MB of RAM and 80 GB disk storage. We indexed a maximum of 100 000 documents from the uct.ac.za domain.

3.2 Indexing

3.2.1 Dynamic versus Static Allocation

The Indexing Dispatcher shown in Figure 1 takes a parameter 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 is because the more 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. Figure 4 shows performance of static versus dynamic allocation. Dynamic allocation was performed multiple times with different reallocation intervals.

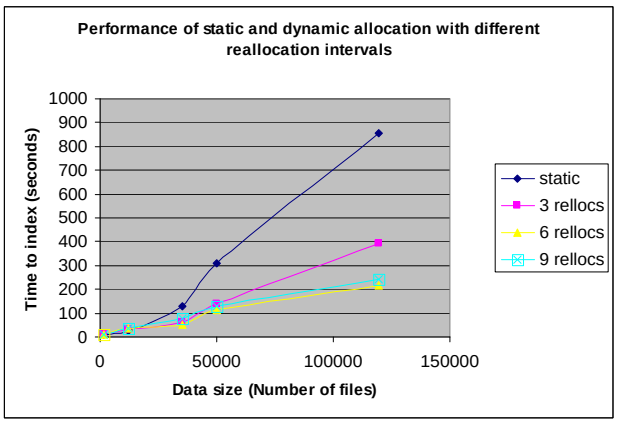

Figure 3: Performance of static and dynamic allocation

The starting number of machines allocated to indexing used in all dynamic allocation scenarios in Figure 4 is 3. Similarly, the number of machines used for static allocation is 3. This is to illustrate that if is it 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 indexing occurs less frequently than querying, there are cases when large amounts of data need to be indexed and indexing becomes more computationally intensive then querying.

From Figure 4, 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.

3.2.2 Cluster Utilization

To illustrate that dynamic allocation achieves better resource utilization than static allocation, the number of idle machines as a function of indexing over querying load average ratio is shown in Figure 5.

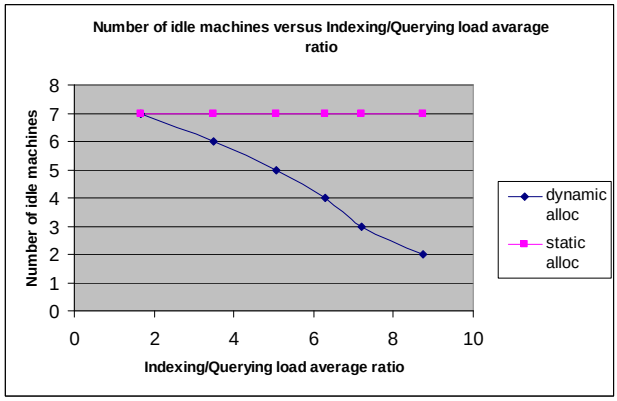

Figure 4: Number of idle machines both with static and dynamic allocation

It is evident from Figure 5 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.

3.3 Querying

To investigate how the cluster is utilized in relation to the dynamic allocation of query jobs, the cluster's load average was recorded against a varying number of queries. As can be seen from Figure 5 the load there is better utilization of the cluster due to the load balancing. As more queries come in the system looks for available cluster machines that a query can be sent to, resulting in an increasing number of machines doing querying.

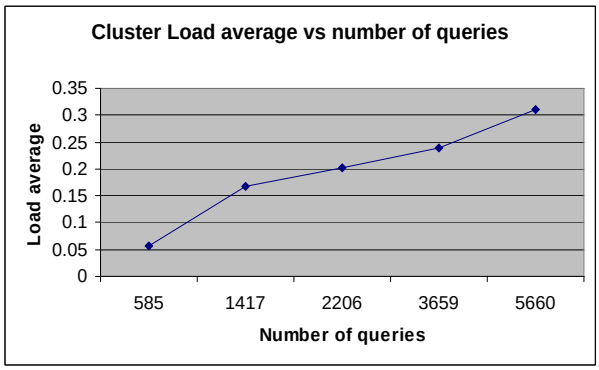

Figure 5: Cluster load average versus number of queries

To investigate the effect of reallocation of cluster nodes 2894 queries were initially run on two machines. As reallocation

occurred, the number of machines doing querying increased and the time to process 2894 queries became shorter. This shows that dynamic allocation of nodes improves the performance of the query processing.

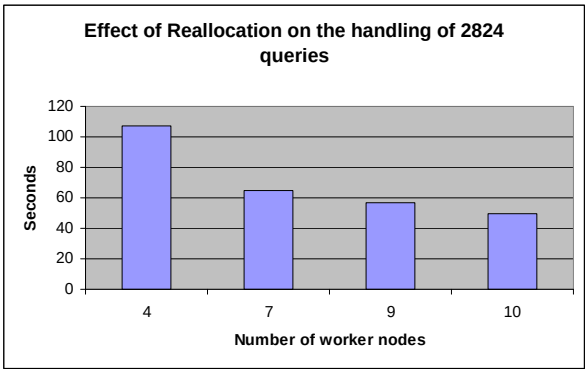

Figure 6: Effect of reallocation on time to respond to queries

4. RELATED WORK

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.

4.1 The Google Cluster Architecture

The Google search engine architecture [2, 6 &7] combines more than 15,000 commodity-class PCs with fault-tolerant software. Each of the PCs 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 [6]. 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.

4.2 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 [7].

4.3 AltaVista, Lycos and Excite Architecture

AltaVista, Lycos and Excite make use of large Symmetric Multi-Processor (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.

4.4 My Own Search Engine (MOSE)

Orlando, Perego and Silvestri [8] 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 parallelization strategy used by MOSE is powerful, and employed by successful search engines such as Google [2], it does not mention anything about keeping the indices fresh.

4.5 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 our system 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 [2, 8, 9 & 10] 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.

5. CONCLUSIONS AND FUTURE WORK

We have presented an approach to better system resource utilization in search engine clusters and discussed how it was implemented in our search engine. We reported the initial results of experiments conducted on a 13 machine cluster. The results highlighted better performance resulting from employing dynamic allocation of querying and indexing to cluster nodes. In particular, we found that for smaller 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, dynamic allocation performs significantly better than static allocation.

There are a lot of important issues that can be further investigated to improve on the solution presented in this paper and to experiment with more varied and larger data sets. Possible future work involves incorporating fault tolerance into the system. Furthermore, to enable the results to be generalized to general applications, the experiments need to be conducted with data from different domains.

6. REFERENCES

[1] A. Ntoulas and J. Cho. What is New on the Web? The Evolution of the Web from a Search Engine Perspective. In *Proceedings of the 13th International Conference on World Wide Web*, New York, NY, 17-22 May 2004, pp.1-12.

[2] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1-7):107-117, 1998.

[3] C. S. Yeo, R. Buyya, H. Pourreza, R. Eskicioglu, P. Graham, and F. Sommers. Cluster Computing: High-Performance, High-Availability, and High-Throughput Processing on a Network of Computers. In *A. Y. Zomaya, editor, Handbook of Nature-Inspired and Innovative Computing: Integrating Classical Models with Emerging Technologies*, Chapter 16, pp. 521-551, Springer, New York, NY, 2006.

[4] GNU Wget. Available from http://www.gnu.org/software/wget/ ; accessed 15 July 2006.

[5] Porter Stemming Algorithm: Available from http://www.tartarus.org/martin/PorterStemmer/; accessed 25 July 2006.

[6] L.A. Barroso, J. Dean, and U. Holzle. Web search for a planet: The Google cluster architecture. *Micro, IEEE*, 23(2):22-28, 2003.

[7] B. Choi and R. Dhawan. Distributed Object Space Cluster Architecture for Search Engines. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, Beijing, China, 20-24 September 2004, pp. 521-525.

[8] S. Orlando, R. Perego and F. Silvestri. Design of Parallel and Distributed Web Search Engine. In *Proceedings of the 2001 Parallel Computing Conference*, Naples, Italy, 4-7 September 2001, pp.97-204.

[9] M. Lifantsev and T. Chiueh. Implementation of a Modern Web Search Engine Cluster. In *Proceedings of USENIX Annual Technical Conference*, San Antonio, Texas, 9-14 June 2003, pp. 1-14.

[10] K.M. Risvik and R. Michelsen. Search Engines and Web Dynamics. *Computer Networks*, 9(3): 289-302, 2002.