NTNU

Norwegian University of Science and Technology

Redistribution of Documents across Search Engine Clusters

Øystein Høyum

Master of Science in Informatics
Submission date: June 2009
Supervisor: Svein Erik Bratsberg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem description

The assignment's goal is to evaluate different methods of distributing and redistributing documents across a cluster of computers with the intention to be used in a search engine. The main focus should be on how these algorithms repartition the documents when the cluster scales to different sizes. The data that should be analyzed are the balance of document's distribution, indexing/lookup efficiency and document transport volume. The assignment should include a theoretical study of the various methods, while the second part should implement the methods on a cluster and see if the theoretical results add up in practice.

Abstract

The goal of this master thesis has been to evaluate methods for redistribution of data on search engine clusters. For all of the methods the redistribution is done when the cluster changes size.

Redistribution methods that are specifically designed for search engines are not common, so the methods compared in this thesis are based on other distributed settings. This is from among other things distributed database systems, distributed files and continuous media systems.

The evaluation of the methods consists of two parts, a theoretical analysis and an implementation and testing of the methods. In the theoretical analysis the methods are compared by deduction of expressions of performance. In the practical approach the algorithms are implemented on a simplified search engine cluster of 6 computers. The methods have been evaluated using three criteria. The first criteria of evaluation are how well the methods distribute documents across the cluster. In the theoretical analysis this also includes worst case scenarios. The practical evaluation compares the distribution at the end of the tests. The second criterion of evaluation is efficiency of document access. The theoretical approach focuses on the number of operations required while the practical approach calculates indexing throughput. The last area of focus examined is the document volume transported during redistribution.

For the final part of the comparison of the methods, some relevant scenarios are introduced. These scenarios focus on dynamic data sets with high frequency of updates, often new documents and much searching. Using the scenarios and results from the method testing, we found some methods that performed be better than others. It is worth noting that the conclusions are for a given the type of workload from the scenarios and the setting for the test. Given other situations, other methods might be more suitable.

When concluding our results we found, for the give scenarios, the best distribution method was the distributed version of linear hashing (LH*). The results from the method using hashing/range-partitioning also showed to be the least suitable as a consequence of high transport volume.

Preface

This report is written as part of the Master of Science in Informatics program at the Department of Computer and Information Science at the Norwegian University of Science and Technology (NTNU). The assignment was carried out from January 2008 until June 2009.

I would like to thank my supervisors Svein Erik Bratsberg and Øystein Torbjørnsen for their many ideas, valuable feedback, motivation and discussion.

Trondheim 01.06.2009
Øystein Høyum

Table of Contents

List of Figures

List of Tables

1. Introduction

The amount of information modern computer systems handle today is huge, and it will keep on growing. When designing a search engine to work on large document collections, it is important to take into account that it must handle this growth. While the performance of computers grows with approximately doubling the processor capacity every 18 months, according to Moore's law, the work created by the amount of documents is too large to handle by a single computer. Because of this a modern search engine system consists of multiple computers working together to handle all the data. The data is split into not overlapping subsets, so that each computer in the cluster gets one subset. To properly utilize the storage servers, it is vital that each subset is of equal size. At the same time, do the uses of rapidly growing data collections require the clusters to easily change size and while keeping the same level of performance. To be able to smoothly handle rapidly growing data collections and dynamic clusters, there is a need for algorithms that are able to properly utilize the capacity of all the computers in a cluster, and capable of handling the change in size of these clusters.

1.1. Goals

Today's search engines handle data sets that are updated frequently. This means that finding the location of documents is an essential and often performed task. It is therefore vital that finding the location of documents is done using minimal number of operations. This is accomplished by keeping an overview of which servers handle which documents. The usual approach to these tasks is to have a master node that stores this information in a directory. Every request for a document must in this case go through the master server. With heavy workload this can lead to the master server becoming the bottleneck of the system. An alternate approach is to use unique properties of the documents to determine which server it is located to. With this approach one can use functions to calculate the location, based on these properties, and get the right server quickly. Important characteristics of these functions are that they must be able to find the right server with very few operations; they must manage to spread the documents equally over all servers independent of the organization of the document input. Furthermore, when the size of the cluster changes, the function must be able to adapt gracefully to the new size, calculating new locations for the documents to utilize the new resources. At the same time the reorganization must not lead to much unnecessary moving of documents. Ideally, only documents moved to new servers should change location.

There are a number of different approaches to distributing data across storage units, and many of them have different backgrounds and are designed for different settings. The goal of this thesis is to examine and study some of these methods. The methods will be analyzed with focus on quality of data distribution, efficiency of document lookup and volume of data transported during redistribution. The evaluation will to include a theoretical analysis and practical testing.

2. Background

This chapter will describe some of the elements in a search engine. This includes how information retrieval works and description of search engine components, the index, compression and the physical architecture of search engines. This chapter will also define some concepts used throughout this report.

2.1. Information Retrieval

Search engines are a type of information retrieval system. This basically means that documents are indexed and retrieved based on document content, and not based on a collection of fields like in database systems. The document content is parsed and the system extracts a collection of words, which acts as a representation the semantics of the document. The words in queries are then matched against these words that act as document representations. (Zobel and Moffat, 2006) describes a query for search engines as a bag of words. Most search engines are using retrieval models like Boolean and Vector model, or a combination to compare the query with the index terms for the collection.

2.1.1. Boolean Queries

The Boolean retrieval model is quite simple. Using the traditional Boolean operators OR, AND, NOT, the query is compared with the terms for each document. A document is retrieved if it matches the Boolean query. Every retrieved document is defined as relevant, but it uses no degree of relevance. The Venn-diagram below shows an example of how a Boolean query acts.

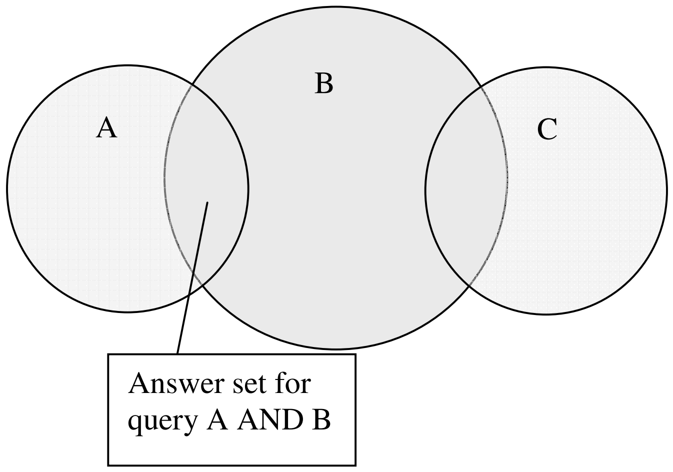


Figure 2-1 A Venn-diagram for the three sets A, B and C

The Venn-diagram shows 3 sets called A, B and C. In an information retrieval setting, this would be sets of documents where set A contained documents with term A, set B contained term B and C contained term C. As the figure shows, these sets can overlap. This means, that in this example some documents contain both term A and B and some contain both B and C. If one performed the query A AND B on these sets, the documents in the area where A and B overlaps would return as the figure shows. The figure also shows that a query with A AND C would return nothing.

2.1.2. Vector space model

Using the vector model, a search engine opens for the possibility of ranking results. Each document and the queries are represented as vectors with a dimension equally the number of unique index terms. For each term in the vector, a set of statistical values are being used to calculate the weight the term as to the document. The weight is usually calculated using a version of the tf*idf function. Here tf is term frequency in the document, and idf is the inverse term frequency. There are a number of ways to calculate these values, but the main goal is to increase weight of terms that has a discrimination function, i.e. it helps identify the difference between documents. Using term weights the vector model compares the documents by finding the angle between the vectors. This is calculated with the cosine function. The cosine of the vectors is found using the following equations taken from (Zobel and Moffat, 2006)

Weight for term t for query q:

$$w_{q,t} = \ln\left(1 - \frac{N}{f_t}\right)$$

where N is the number of documents, and f_t is the number of documents containing term t

Weight for term t for document d:

$$w_{d,t} = 1 + \ln f_{d,t}$$ where $f_{d,t}$ is the frequency of term t in document d.

Vector for document d:

$$W_d = \sqrt{\sum_t w_{d,t}^2}$$

Vector for query q:

$$W_q = \sqrt{\sum_t w_{q,t}^2}$$

Using these variables the cosine formulation is used to calculate the similarity as follows

$$S_{q,d} = \frac{\sum_t w_{q,t} \times w_{d,t}}{W_q \times W_d}$$

The result will be ranked by degree of relevance. This also means that you can find documents that do not contain all the query terms.

2.1.3. Alternate models

A system that introduced a bit different strategy, then the one mention above, is the Google search engine. Google is also using the traditional comparison models, but it also includes the PageRank algorithm (Brin and Page, 1998) and merges the PageRank results with the other results. PageRank is a static ranking strategy. This type of ranking does not depend on the query. The ranking in PageRank is based on the number of links going to and from a webpage and to what other pages the webpage is connected to.

2.2. Search Engine Components

(Risvik, 2004) defines that a search engine consists of four major parts; the crawler, the indexer, the searcher and the dispatcher.

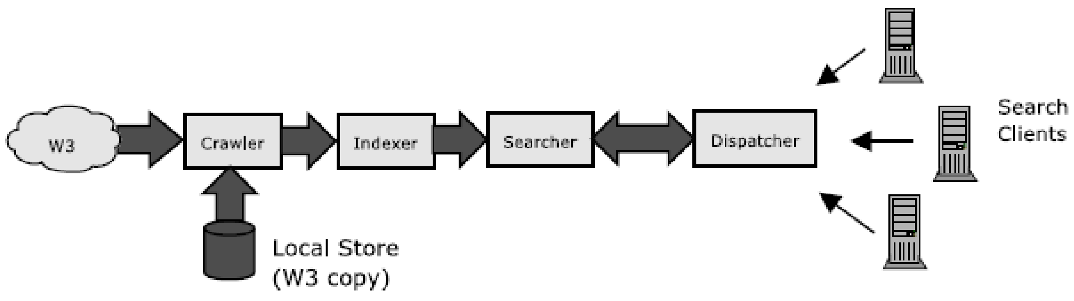


Figure 2-2 The connection between components in a web-search engine. Illustration taken from (Risvik, 2004)

2.2.1. The Crawler

The role of a crawler is to retrieve information from documents from a document set, like the web or an internal document collection in an organization. This information is what is used by other parts of the search engine to index the document collection. In a collection that is connected by hypertext, the process of crawling starts with a set of URL's. The crawler then follows every link possible from these pages, until every possible document is reached. The information collected is then sent to the indexer. The crawling process is usually done regularly, but not a total crawling process is necessary each time. This is because the difference in update pace for different pages. For web pages, a page belonging to a newspaper is updated several times each day, and has major updates from day to day. Homepages for corporations and organizations have a quite different pace of update, and therefore does not need to get visits from a crawler as often.

2.2.2. The Indexer

The indexer uses the output from the crawler and creates a searchable index for the document collection. There are a few different types of indexes that are being used: Inverted files, suffix arrays and signature files. The best and most used, by far, is the inverted index. Using compression this gives the best combination of speed and storage requirements. An inverted index is similar to a word index that you find in a book.

2.2.3. The Searcher

The searcher is the module that does the actual work of finding the answer for the documents. Usually a search engine is a quite big system, which is distributed over several computers. Because of this, the search engine must run a searcher on every computer. The searcher has access to an index for the server it resides on. After the query processing, the result of the search is sent to the dispatcher

2.2.4. The Dispatcher

The dispatcher accepts queries from users. The first job of the dispatcher is to distribute the query to the different computers in the search engine, and the searchers. After this the dispatcher receives answers from the searcher. These part answers are then merged and then returned to the user.

2.3. The Index

An index is an essential part of any system with as much data as a search engine, and maybe at a larger degree than other systems, since it is mostly unstructured data. The job of an index is to let the system make an effective search on the data material. There are various types of indexes possible to use in search engines.

Among them are the inverted index, suffix structures and signature files. The inverted index consists of a vocabulary of all the index terms in the collection, and for each term, an inverted list. The inverted list consists of a pointer to every document which contains that term. The list also includes the number of occurrences of the word in the documents, and often the word positions also.

Suffix structures store all possible suffixes of the text in the index. You can use either a suffix tree or a suffix array. In most cases the suffix array is the most appropriate, since it takes less space. This structure has the possibility to perform a wider range of queries than an inverted index, but it has the drawback that needs a lot of space and compression is not possible. It is also a time consuming process to create these kinds of structures.

The third index structure is the signature file. This uses hashing to create a bit signature for all the terms. By using bitwise OR on all the terms in a document, a document signature is created. To check if a document contains a term, the signatures are compared, and if the term signature is a subset of the document signature, one might have a match. This method can give a false match, so a retrieved document has to be post processed before sending it as an answer to the user.

The inverted index is according to (Zobel et al., 1998) said to be the best type of index, this is because of good speed and compression, and is therefore the most used index. There are of course some systems that use the other methods. For example, according to (Risvik, 2004) the index used for the FAST search engine is a version of the suffix array. Since the inverted index is counted as the best and most widespread index, for the rest of this paper an index will be counted as an inverted index.

2.4. Compression

Central to the development and working of a search engine is the use of compression. Most search engines have to handle huge amount of data, and it is therefore natural to try to reduce this as much as possible. There are two types of compression in use in a search engine. That is compression of the actual documents, and compression of the index.

2.4.1. Document compression

The compression of documents in search engines mostly use text compression, and this will therefore be the main focus of this sub-chapter.

Text compression is as a rule done lossless. This is natural as it can not be tolerated that letters are missing from text after decompression. The use of lossy compression is on the other hand the normal procedure for other types of data, like images and audio, because this has no effect for the average user. When using lossy compression on these kinds of documents, the data that is lost during compression is of a type that a human is incapable to perceive.

Some of the most relevant methods of lossless text-compression are symbol-based methods like Huffman- and arithmetic-encoding, and Ziv-Lempel a dictionary-based method. The symbol-based methods use models that extract the predictability that given symbols will appear in the text to be compress these symbols. Using these models the methods code the symbols in the text so that in average each symbol uses less then the uncompressed 8 bits. To find the statistics of the text the models either can do a text scan before the coding stars, or use an adaptable conduct. The adaptable method starts with a default statistics of the symbols, and as the text is traversed and encoded, the model change to a more correct view if the text. This latter way of creating a model is the one most used in today’s systems.

Huffman coding was developed in the early 1950s by David Huffman(Huffman, 1952). This encoding was among the earliest and best compression algorithms until the Ziv-Lempel algorithms where published in 1977 and 1978, and arithmetic coding was first granted patent in 1978 (Glenn George Langdon and Rissanen, 1978). The principle of compression using Huffman coding is to assign a code with few bits to symbols appearing very often in the text, and longer codes to rarer symbols. This way the most used symbols have few bits that are less then the 8 bits of uncompressed text, while the less used symbols might get more than 8 bits. The codes for the symbols are stored in what is called a Huffman tree. A modification of Huffman coding is something called canonical Huffman coding. This type of coding replaces the codes in the normal Huffman tree with alternate codes to accelerate the decoding speed.

Arithmetic coding bases the coding on models, just like Huffman coding, but instead of replacing the codes for each symbol, arithmetic coding makes a representation of the whole text as a single decimal number. Encoding with arithmetic coding bases itself on the probability distribution of what the next encoded symbol is going to be. The set of probabilities creates a set of intervals from 0 to 1. When one symbol is encoded, the probability interval of this symbol is again split into intervals after the probabilities for the next symbol. This process will continue creating a decimal number with greater and greater precision.

The difference of these to symbol based methods is according to (Witten et al., 1999), that arithmetic coding gives the best potential compression, close to the best possible, but a bit slower than Huffman coding. For use in search engines and information retrieval system Huffman coding is often preferred for text retrieval, as it is more suitable for random access of the text. For images, arithmetic coding is used, since random access to the image content is not of particular importance. Compared to dictionary based methods like Ziv-Lempel, symbol based methods require a lot more memory. This is because symbol based methods needs to store the probability models. Huffman coding without canonical codes has even greater memory requirements, since it needs to store the Huffman tree also.

Among dictionary based methods the most known are the one based on the work of Jacob Ziv and Abraham Lempel(Ziv and Lempel, 1977) and (Ziv and Lempel, 1978). The main principle behind these methods is compression of symbol and substrings representation, by creating pointers to substrings previous in the text with the same content. Since the dictionary used the text itself, there are no extra data that has to be transmitted, and it does not require the same amount of memory as the symbol based methods.

2.4.2. Index compression

While the inverted index has the best storage utilization of the mentioned indexes, it still uses quite a lot of space, when used for data collections such as those from search engines. It is therefore a key issue to find a good way to compress the index as much as possible. One simple way is to compress the integer values in the index. For every inverted file, document identifiers are stored in sorted order. The identifiers can be stored more efficiently by storing the delta values between them instead of the whole value. To further compress these values, they can be encoded in various ways. The normal way to store an integer is to use a constant number of bits that can handle a wide range of values. This is a waste of storage, when a lot of the values are rather small. The options of using a smaller number of bits for all the values are not a possibility, since some values can require a larger number of bits. The solution is to use an encoding that varies the number of bits. The most efficient coding is the Golomb-encoding that are based on the Benoulli-model. The Golomb code uses parameters which based on the probability that a random word is found in a document. This parameter is then used to compute the two parts of the Golomb-code. The first part is then coded in unary format, and the other in binary format.

2.5. Definitions for expressions in the text

2.5.1. Node

The term node can be seen as an abstraction. This is because physically there can be multiple nodes on a single computer, or alternate a node can consist of multiple computers. In search engines and computer networks a node is a unit with clearly defined responsibilities and work areas. The types of nodes can for example be computation or storage nodes. Search engine nodes are usually either searching nodes, client nodes or master nodes. Search nodes will do the actual query processing and storage of documents and is therefore the main node type. Furthermore, there are client nodes that retrieve queries from external parts of the system, and are responsible of distributing queries to the nodes and process the results from the queries. The last main class of nodes is master nodes. A master node is not necessary in every system. The task of a master node is to manage traffic and keep directories to the other nodes. Most of the cases studied in this thesis do not need a master node, because document distributing is deterministic and therefore there is no need for directories. One additional type of node used in the experiment, presented later in this thesis, is a scaling coordinator. This handles a small part of the job of a master node. The job of the scaling coordinator is to give messages to the nodes to start the scaling process when one of them sends a message that it is full.

2.5.2. Storing of documents

The algorithms examined during this thesis provide methods to map documents to nodes. How documents are stored at the nodes is not something handled by theses methods. The reason for this is connected to the definition of node. As noted earlier nodes are an abstraction. This allows the distribution of documents to be a high level operation. Keeping the nodes structure abstract will make system more robust, because the underlying structure of the nodes can be changed without affecting other parts of a system. Furthermore, with this architecture it is irrelevant for the distribution methods if the nodes run Windows, Unix-based or Mac operative system and which file systems is used. The only responsibility for the algorithms then is to determine which node to store the documents while the nodes themselves do the block allocation.

2.6. Inserting documents

2.6.1. Physical placement

The placement of documents in a distributed setting can be done in two different ways. The first is based on a central directory that keeps records of where each document is placed. The master node that contains this directory also keeps track of the load balance of the nodes to avoid data skew. The other way is based on deterministic placement of the documents. Deterministic document placement uses unique properties from the documents to decide their location. This can either be done by splitting the document collection in partition and place each partition on a node, or determine placement with hashing. The method with range partitioning can be combined with the use of a central node which keeps track of partition borders, or let the lookup jump between nodes until the correct is found. This last method is currently used on the web for distributed hash tables.

When document distribution is determined by hashing, there is no need for a master node, since one just inserts document identifiers into a function, which calculates the right node. In systems like search engines, the node are usually tightly clustered. This means that the nodes know of each other, and therefore are able to send documents to each other after a document address is calculated.

The method described, which uses a master node to determine document placement, is the simplest method and is currently used by Google in their system, but with some performance related modifications. The downside of this method is the system potentially has a single point of failure, and that the master node gets flooded with messages and queries.

2.6.2. Updating the index

A central part of inserting documents is to insert the documents to the index. This can be a rather complex job, since an index like an inverted file has a large number of pointers for each file. The first part of the job is to parse the new document to extract the index terms. Then the vocabulary must be searched for each term in the new document. If a term does not exist, it must be inserted also. Luckily the vocabulary can most often be stored in memory and as a B-tree. Therefore one does not need disk access for this operation, and the searching for all of the terms can be done in O(m*log n) time , where n is the size of the vocabulary and m is the number of terms from the document. This structure reduces the cost a bit, but the most expensive part of the update is insertion on the inverted list. This part is done by accessing the inverted list of every term from the new document and inserting the document id, term frequency and term positions in that list.

2.6.3. Monitoring the system state

To be able to have the best possible performance of the search engine, there are some variables that need to be monitored. These variables are mostly connected to the load balance in the system. During inserting of new documents, the state of each node must be monitored. This usually means to register the storage load on the nodes. When at least one node has reached its capacity, the system administrator must expand the storage cluster, and let the system run algorithms that redistribute the data, so that the new nodes also get utilizes.

2.7. Data structure/ Architecture

The architecture used in most search engine is a distributed structure based on the shared nothing architecture. This means that the nodes in the network are normal computers, each with its own disk, memory and processor. A trend among search engine providers is also to use simple computers as nodes instead of expensive high performance computers. This makes it cheaper and easier to upgrade the network or replace failed node. Also with the large number of computers, the use of high performance nodes does not increase the overall performance much more than with the use of more ordinary computers.

Since the data collection is often very large, it needs to be distributed over several computers. One method of doing the distribution is partitioning based on the terms in the index. This means that each node has responsibility for its own part of the index. In this, the search can be performed on just some relevant nodes. This method is called term partitioning. The other possibility, called document partitioning, is to partition the documents so that each node gets an equal share of the data collection. With this method the answers from every node must be merged to a single result, before it is sent to the user. It is this last method that is currently in use in most systems today. This is because it gives good query throughput; it balances the load to the different nodes very well and has good scalability. Since document partitioning is the method currently most in use, when mentioning partitioning/distribution of documents, later in this thesis the method in question then is document partitioning.

A central element of the data architectures is replication. Since a search engine incorporates many computers (up to 1000 per sub-cluster with Google according to (Ghemawat et al., 2003), probably even larger at present) and large data sets, it is expected that nodes will fail often. It s not a possibility to let the search engine ignore the data belonging to failed node, therefore only option is replication. One way of doing this, is to have some nodes that are identical or mirrored. This makes it possible to not just distribute the documents, but also the queries. If one node is busy, one can just send the query to a mirrored node. Another possibility is to distribute the replicas over all nodes. With this organization a partition does not loose all of its replicas once with a node failure.

3. Data distribution

The amount of data handled by and stored in a search engine is rather large. This means that these documents need to be distributed across a cluster of computers. In our modern society the data available constantly grows, and in the same way the number of documents in search engine grows. Because of this it is not possible to make correct estimation on the required size of storage clusters. It is therefore essential that the system uses algorithms that are able to handle the growing and the utilization of a number of nodes in a way that does not require a too extensive reorganization of the data.

One of the key issues here is that the algorithms must distribute the data to the servers, so that every node gets at equal load. How this is done is also dependent on the type storage cluster. In homogeneous clusters where all nodes have the same capacity, the search engine must distribute the documents equally over all nodes. The other case is heterogeneous clusters where the nodes are of different types. In this situation the distribution must let the most effective nodes get an amount of load equaling the capacity of the nodes.

Without a proper distribution of data the system is at risk of getting nodes which become bottlenecks. In this case there would not be of any use to expand the system, since there would still be capacity not getting utilized.

Later in this thesis, some possible algorithms for data distribution and scaling will be introduced. Most of them determine data placement with the help of hashing or randomization. There are a number of advantages with these kinds of distributions. One of them is that the functions give a balanced data distribution and the deterministic data placement which makes a master node unnecessary. Also hashing usually is a fast way to find the correct node, as opposed to checking a global directory.

To sum up, the requirements for distribution methods are:

- It should be able to distribute documents equally across all nodes
- It should effectively find the correct storage node for documents
- It should gracefully scale in size without restrictions
- It should limit the number of document moved during various operations

3.1. Query load

An important element to consider while distributing data is the effect of query load on the servers. The goal that most of the algorithms strive after is to get this balanced data load, and through that views all documents as equally important. This is actually not the case in most modern search engines. As stated in (Badue et al., 2007) "*even with a balanced distribution of the document collection among index servers, correlations between the frequency of a term in the query log and the size of its corresponding inverted list lead to imbalances in query execution times*". What this means, is that some terms appear in more queries than others, and therefore some documents are more often retrieved that others. In a cluster where the documents are evenly distributed this will therefore lead to some slower performance to some nodes because of increased disk accessed. In real life search engines this is often solved by caching popular documents and queries. In this thesis the focus is on the document load on the servers. These elements will therefore be outside the scope of this thesis. Further in the thesis, the main focus will be on getting equal load of documents and not include the effect queries have on performance.

3.2. Methods of distributing data

This chapter will now introduce the three basic methods of distributing data. Some of these methods are in turn used by the algorithms studied in this thesis.

3.2.1. Round-Robin

Distributing documents with round-robin, places the documents to nodes without any mapping determined by document keys. As documents get in to the system, they are distributed to the nodes in order where each node has its turn. This method lets each node get an equal share of the documents.

Because there is no initial structure determining the placement of documents, the system can during scaling choose randomly which documents are moved to new nodes. This leads to a minimal movement of documents during scaling.

The basic distribution with round-robin places documents equally over all nodes. This is good enough for homogeneous clusters, but if the different nodes have different capacities there might be under-utilization of some nodes. To handle heterogeneous clusters better the round-robin algorithm can weight the nodes differently. Using this weighing the most important nodes can get some extra visits during the distribution.

The drawback of using round-robin is the fact that the loose mapping between nodes and documents forces a system to use a directory to keep track of where documents are. A directory can often become a bottleneck and a single point of failure of a system, since the majority of the request must go through the directory.

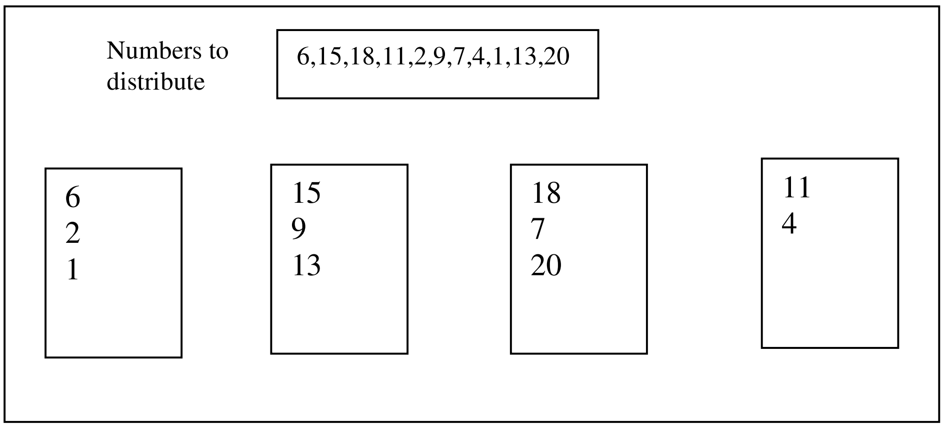


Figure 3-1 A number series distributed using round-robin.

3.2.2. Range-Partitioning

Using range-partitioning, the set of possible document keys are divided into partitions or sub-sets of keys. The number of partitions are equal the number of nodes, so that each node gets its own part of the keys. The documents are then placed on the node responsible for the range of keys which the document key belongs to. This method requires that the document identifier can be represented as or converted into a numeric value.

When scaling a cluster using range-partitioning, the set of possible keys are re-divided, so that the number of sub-sets equals the new number of nodes. The nodes in the cluster then need to change the borders between them, and then move the documents to satisfy the new distribution. The scaling of a cluster leads to a large number of documents that needs to move between nodes. To minimize this number, the new node should not be assigned the end of the key range, but instead be assigned a partition in the middle of the range.

If the cluster is primary a homogenous cluster, the set of keys can be divided into partitions of equal size. When there is a heterogeneous cluster, or if some parts of the keys are more popular, the system can assign partitions of unequal size to get the best performance possible of the cluster.

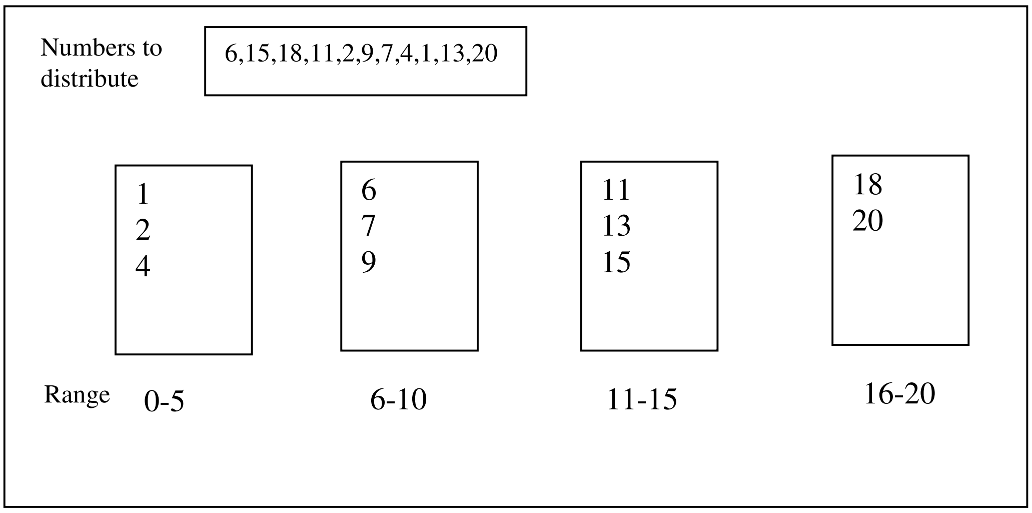


Figure 3-2 A series of numbers distributed with range partitioning.

3.2.3. Hashing

To distributed documents with hashing is basically using a function which determines the correct document placement. The goal of this function is, that based on the range of possible input keys the resulting values shall be equally distributed across the whole result range.

A hash function usually works as follows. The document identifier is converted into a numeric form, e.g. the ASCII of the string. In search engines this identifier is usually in the form of an URL. The number is then multiplied with a large prime number, and the resulting number is represented as a 32-bit or 62-bits integer. The large set of possible values and the multiplication enforces the dispersion of the results. When the 32-/64-bits integers are to be mapped to a much smaller range of values, a modulus function is used with the 32-/64-bits integer. This converts the large numbers to a smaller numbers without loosing the distribution among the possible values. When used on a cluster of nodes, the modulus maps the 32-/64-bits integer to the number of nodes in the cluster.

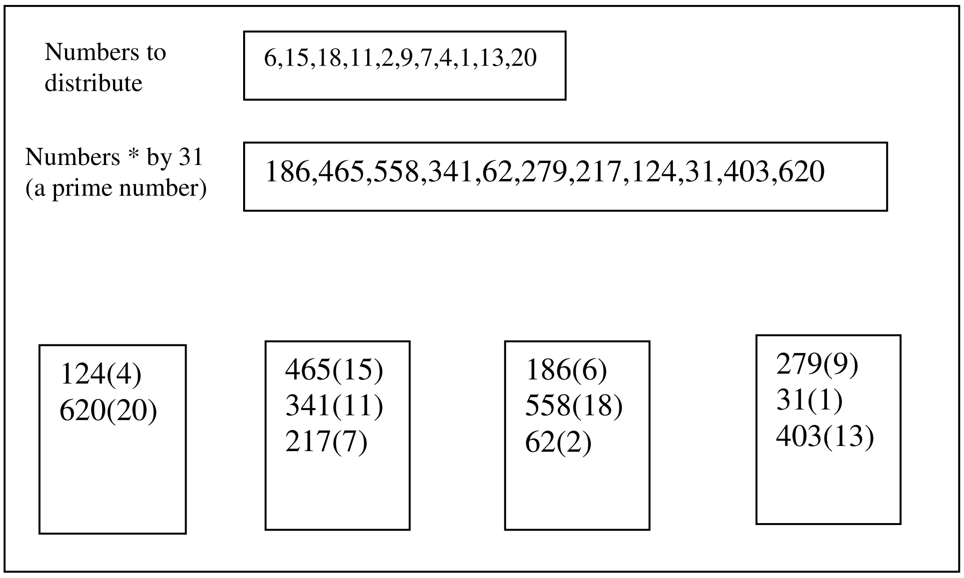


Figure 3-3 Example numbers are multiplied with a prime number and how the multiplied numbers are distributed using modulus function. The prime number used is rather small to simplify the example.

When using hashing to distribute documents across a cluster of nodes, each node gets an equal part of the documents independent of the document identifiers used for input. A problem arises when the number of nodes changes. This leads to the use of a new modulus function. The new function will most often change the placement of a majority of the documents, and extensive document movement is something that should be avoided. Some of the algorithms studied later in the thesis are hashing methods that deal with this problem by enabling hashing that scales gracefully, i.e. a new functions just leads to a minimal number of documents moved and not a complete reorganization. Among the most well known algorithm of this type is linear hashing and extendible hashing.

As with round-robin, the basic distribution with hashing does not differ between nodes of different capacity or generation. With some modifications the hashing can also handle heterogeneous clusters. The document keys are hashed to a set of buckets using a modulus-function. If the number of buckets are changed from equal the number of nodes, to a grater number, then multiple buckets is placed in each node. To handle heterogeneous clusters better, nodes with higher capacity gets more buckets than lower capacity nodes.

4. Data distribution algorithms

4.1. General considerations

Before describing the actual algorithms, the general setting for the algorithms will be presented.

4.1.1. Setting

- The algorithms are used to distribute documents across a cluster of N nodes.
- Throughout the life-cycle of the cluster number of nodes N will change, either be smaller or bigger
- The documents are identified by a unique pseudo-key X, which is created based on the URL of the document.
- The type of document is generic. It can be all kinds of data types that search engines typically handle.
- The load on the nodes does not include query load but storage load.
- The algorithms are primary going to be used when deleting, inserting or updating documents.
- For the algorithms specific hashing function or pseudo-random number generator are not described. There are several possible algorithms publicly available which can be used for implementations. There are e.g. functions available thought the API of various programming languages. In the analysis part, there is assumed that suitable algorithms for hashing and random number generating are chosen that gives satisfying performance.

4.1.2. Operations

The operations performed on the cluster are

- *Insertion*: The process when a document is sent to the cluster by an external component. The document is either received by one of the storage nodes or a client node. This depends on which algorithm is used. Where the document is stored is calculated based on algorithm definitions, and the document is sent to the correct node.

- *Deletion:* Deletion is basically the reverse process of insertion. When the cluster receives order that a specified document is to be deleted, the same algorithms are used to calculate where the document is supposed to be. If the document exists, it will be deleted.

- *Update:* For this study, an update is performed as a deletion followed by insertion and not in site updating.

- *Document redistribution:* This are most often performed when the cluster is scaling in size. Each node has its own version of the distribution algorithm and an image of the cluster. When the nodes get an update on the status of the cluster, it iterates for every document it stores and calculates the placement for them. If necessary the documents will be moved to a new location.

4.2. Linear Hashing (LH*)

LH* (Litwin et al., 1996)is a further development of the original linear hashing algorithm (Litwin, 1980), and is specified to be used on distributed systems. The * is used to indicate this difference.

LH* uses two hash functions h_i and h_{i+1} to map objects to nodes in a cluster. The index *i* is an indicator of the "level or "stage" of the algorithm, and tells how many times the cluster size is doubled. A usual implementation of h_i is to place an object to node

$$h_i = X \bmod N*2^{i.},$$

where N is the original size of the cluster and X is the object key or a pseudo key. LH* expands the cluster by splitting one of the nodes. Splitting is done first after the cluster reaches a minimum limit of storage utilization. After this limit is reached and one of the nodes is full, the splitting of nodes starts. When the storage cluster is expanded and one of the nodes is split, its content is moved to the new node, or more precise node

$$n+N*2^i$$

Which node to split is indicated by a split pointer n. The pointer n is pointing at the leftmost node that uses the h_i function. When a node is split, this node and the new node starts using the function h_{i+1} for document placement.

When all the $N*2^i$ nodes have been split the level is incremented to i=i+1, and the hash functions are updated so that

$$h_i = h_{i+1} \text{ and } h_{i+1} = h_{i+2}$$

To make the LH* algorithm as scalable as possible, there is no use of a master node to organize the distribution of objects or system expansion. Each storage node and each client communicating with the cluster has its own versions of the system level variable and split pointer. These are not updated throughout the system when the system expanded. It is therefore possible to gets inconsistent values. Because of the way the algorithms works, the inconsistency is never more than one over or one under the actual value. This means that the maximal number of false lookups is 2.

In the years after LH* was introduced, several new versions has been published that are specializations of LH*. This includes $LH*_M$, $LH*_g$, $LH*_S$, and $LH*_{RS}$. These new developments incorporate different functions for high availability and security. This master thesis will primary concentrate on the basic LH* algorithm for distributed systems, since the thesis focus on data placement.

The biggest drawbacks for the family of linear hashing algorithm are that the expansion of the cluster is done by splitting one of the nodes. This operation means that in a system with roughly equal load, the split node and the new node will have approximately half the load of the rest of the cluster. This means, that only when the number of nodes is a power of 2 multiplied with the original number of nodes, the system has a perfect load balance. Another drawback is that splitting only happens when the system load is past a boundary, and not when a node is full. Because of this the full node must handle overflow until it is this nodes turn to split. It is also not necessarily the full node that will be split and it might be a node with very little node. These things affect the load balance in the system, and preventing it from being ideal.

4.3. Extendible Hashing (EH)*

Just like the LH* algorithm, the EH* developed by (Hilford et al., 1997) is a specialization of an earlier extendible hashing algorithm (Fagin et al., 1979) for dynamic files modified to use in distributed systems.

The basic principle of the extendible hashing algorithm is making a table from a binary search tree. The original algorithm consisted of two parts: an array of addresses and the storage nodes. Extendible hashing is using the same hash function over the whole system lifecycle. The hash function distributes values uniformly over a large range of values. The algorithms use a global variable called the system depth. This indicates the number of addresses available in the array. For example, if the depth is 3, the number of addresses is $2^3=8$. In this way the depth also tells how many bits the addresses are going to use. With a depth of 3, it is the three most significant bits inn the key which decides what addressbucket it belongs to. The second layer, containing the nodes, has pointers from the addresses where each node may have multiple addresses connected to it. When a node has reached its storage capacity the node is split. At this time a new node is inserted to the system, and the new node gets half the data from the node that split. At the same time the pointers to the old node is shared between the two nodes. If the system is in a state where the full node only has one pointer, the depth variable must be incremented to create more addresses. These are then distributed to the nodes.

The problem with the original algorithm that did not support the use in scalable settings was the use of a central directory. This can be a bottleneck if it receives many requests, and it will eventually grow quite large. In (Hilford et al., 1997) this is solved by using what they called cache tables. Cache tables are versions of the directory distributed across all the nodes, both client and servers. Since a system using EH* are going to be able to scale to very large number of nodes, it is not a usable option to update all the cache tables for each insertion or deletion. The solution from (Hilford et al., 1997) introduces is a kind of lazy update. The lazy updates make it possible for the system to have different values for the depth variable in different nodes. EH* solves this by letting each node store borders for the max and min values for the system depth. When an insertion is performed, the client will search its cache table for different values of the system depth from the min value to max value until it finds a node to send the documents to. Further, it is a possibility that the cache tables in the clients are obsolete. In this case the node that receives a document but is not the correct one, it will use its own cache table to find a correct node. This process is repeated until the right one is found.

One problem that arises using this decentralized method is that the multiple possible values of the depth variable make the lookup slow. This would be equal to traversing a binary search tree from root to leaf. The servers or clients needs to search its cache tables using all their possible values for the depth, and further the resulting node might be wrong leading to a chain of forwarding a document because the depth variables is obsolete.

The figure bellow shows how this sort of problem might arise in a EH* cluster. The system initially contains two clients and a server A. As long as the initial server has free capacity, every key is mapped to this node. In this case, client 1 has trigged the splitting of the initial server, and every key ending with a 1 bit is sent to the new server. This new mapping information is contained in the two servers involved in the splitting and the client that triggered the split. The second client is not yet updated and will therefore send every object to the same server. This information will first be updated when this client send an object that should be on server B to server A.

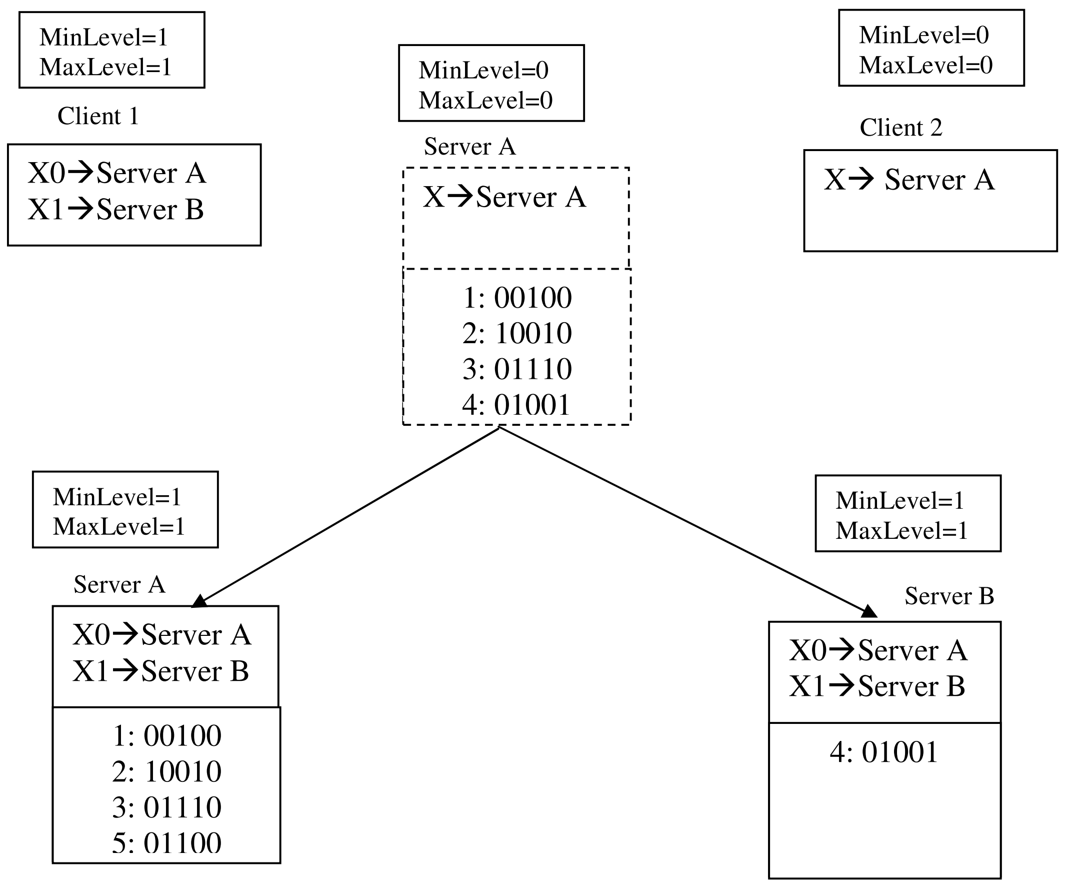


*Figure 4-1Example of node splitting with EH**

4.4. RUSH

RUSH or Replication Under Scalable Hashing, is a family of algorithms developed by (Honicky and Miller, 2004) that incorporates data distribution, replication and node weighting in the same algorithm. There are three different algorithms, $RUSH_P$, $RUSH_R$, and $RUSH_T$. The difference between the RUSH algorithms and the other algorithms analyzed in this thesis, is that RUSH includes how to replicate documents in the algorithm, as opposed to the usual way where replication is done by mirroring a node. A second difference is that RUSH distributes documents on the basis of heterogeneous clusters instead of homogeneous ones.

One central element the RUSH algorithms share is that they make the assumption that disk are added to the system in clusters, and also that the nodes in a cluster is of the same type. Servers which are inserted at the same time, but are not equal, must be added in separate clusters.

The clue behind the notion of grouping disk this way is that the system then knows where to find the newest disk, as these will typically have better performance then older disk. The clusters will usually receive weights based on performance and age. The algorithms use this knowledge and will always try to map new documents to the newest disks. The algorithms use recursion and will start trying to map documents to the most recent added cluster. If this is not done, it will try the second newest cluster until the document is mapped to a cluster, or it reaches to oldest cluster. This might seem like a time consuming process if the algorithms risk traversing all the clusters. This will rarely happen, because most documents will usually end up at the newest cluster. If this is to be the case it is also essential that as new clusters are added, these will have better performance then the older. If the oldest cluster is the highest weighted cluster, most documents will then end up there.

The different RUSH algorithms have its own way of mapping documents to clusters and nodes. The first RUSH_P incorporates the use of prime numbers to find the correct node. RUSH_R uses a hypergeometric distribution and RUSH_T use a three organization.

The following figure shows how a cluster might distribute 1 million documents after different number of scaling reorganizations. The columns represent sub-clusters with 5 servers each. As the figure shows, the number of objects pr cluster increases roughly linear as the clusters becomes newer.

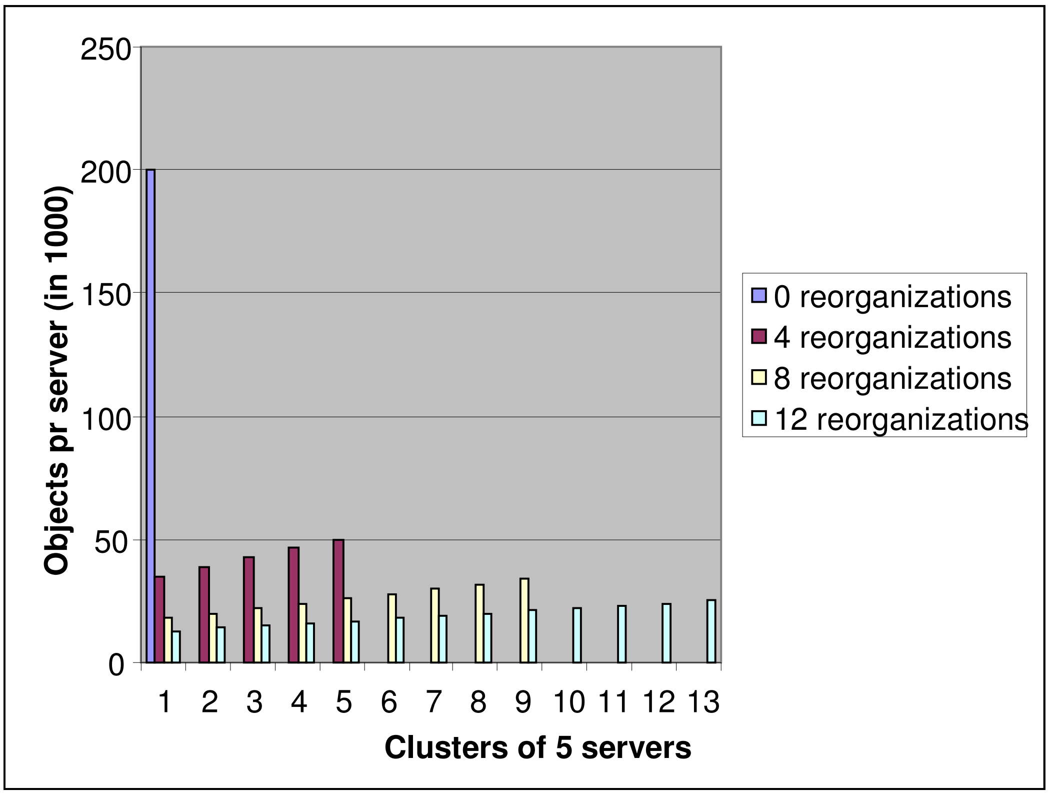


Figure 4-2 How the RUSH algorithm distributes objects

4.4.1. RUSH$_P$

To get a good load balance among the servers, the document keys need to be randomized. RUSH$_P$ is doing this by using the key as a seed in a random number generator. The system keeps track of the number of clusters *j* in the system. This variable is then used to determine how many iterations the random number generator shall perform. The number found is normalized so it is always less then the total weight of all the clusters in the system.

To determine if a document belongs to the most recent cluster, the algorithm compares the pseudo key with the weight of the new cluster. If the pseudo key is the smallest, this will be the right cluster. To find the right server the algorithm, uses a regular modulus function with the pseudo key and server numbers as input. If the pseudo key has a bigger value then the cluster weight, the whole process is repeated for the next cluster.

The description above is the most basic way the algorithm distributes documents. When the placement of multiple replicas is taken into account, each replica gets its own number. A new pseudo key is generated for each replica that consists of the old object pseudo key, the replica number and a random prime number. The role of this prime number is to make sure that none of the replicas is placed on the same server. To state the obvious, the whole point of replicas is to put them on different servers in case of server crash.

4.4.2. RUSH$_R$

Just like the previous method this is using the weight of the most recent sub-cluster and the weight of all clusters in the system to determine where to store a document. The difference is that this method finds the correct node for all of the replicas simultaneously, instead of in multiple operations as the RUSH$_P$. Using the object key and cluster weight as parameters to various functions, the algorithm finds the correct node. The first step is to find out the number of replicas which are going to get stored on the newest cluster. This is done by a function that is using the hypergeometric distribution. A hypergeometric distribution is a probability distribution describing the process of drawing a number of draws from a population without replacement. For each document there is defined the number of replicas that are to be inserted into the system. If the number of replicas the hypergeometric function found is less then the total number of replicas, this procedure is continued for the next cluster, just like the previous RUSH algorithm.

When the number of replicas is found, the algorithm must distribute these across the cluster. This is done by another statistical based function. This function finds k elements from a set of n integers. The selection is done without replacement.

One of the main goals when RUSH$_R$ was developed, was that it had to manage the removal of servers and not just adding new servers. RUSH$_R$ does this by doing removal as a kind of reweighting. To remove a server, the administrator just has to set the server weight to zero, this will then lead to movement of all the documents on that server. There is one drawback on this method: more than the optimal number of documents will be moved when removing servers. If one old server is removed, most of the newer servers will change their weights also.

4.4.3. RUSH$_T$

The RUSH$_T$ differs from the previous two by organizing the clusters as a tree rather as a list. The first operation to find the right sub-cluster is done by traversing the tree from root to leaf. A pseudo-key is calculated for an object. It is then compared with the three node identifier. If the key is smaller then the identifier the node is sent to the left child node, else to the right node. This process is repeated until the algorithm reaches a child node on the three. This is where the sub-clusters are stored. When the right cluster is chosen, the RUSH$_T$ uses the same method as RUSH$_P$ find the right node.

In this thesis the focus will be on the RUSH$_R$ algorithm. This version has a good combination of easy implementation and good performance compared to the other two alternatives.

4.5. SCADDAR (Scaling Disks for Data Arranged Randomly)

The SCADDAR algorithm (Goel et al., 2002) was developed for use in systems storing documents of continues media, like audio and video. In SCADDAR the documents are divided into blocks. The algorithm uses a function generating a series of pseudo-random numbers based on a unique seed for each object. The function iterates *i* times each returning a random number, where *i* is the number of blocks for an object.

When a pseudo-random number is generated for a block, it is mapped to a server using a modulus function.

$$S = X \bmod N,$$

where S is the server X is the pseudo-random number and N is the number of servers. When the system is scaled, the algorithm rehashes the blocks, but only moves blocks that belongs to a new server. When the scaling reduces the number of servers, it uses the similar approach where new addresses are calculated for blocks belonging to the servers that are to be removed. These are then mapped to the remaining servers. With this approach the algorithm only moves the minimal number of nodes.

SCADDAR scales the system by using a series of randomization functions. When a new node is inserted to the system, a new function for random number generation is created for this the new number of nodes. The next step is to calculate which objects that are going to be stored at the new node, and move these objects. For documents not stored at the new node, the placement is still determined by the old random number function. With this method, the system does not need to move as many objects among old nodes as a complete rehashing would require.

The randomizations functions calculates a new pseudo key X used as input in the function listed above. By calculating new a value of X it is determined if a document is mapped to a new or an old node. The function calculates two possible types of output. The function might calculate an X, which will map the document to a new node. This will then be the output. The alternate output will be the X having a value determining that the document is to be stored on the same node, which the document currently resides.
In (Goel et al., 2002) three different object mapping functions are defined. The first is a naive method that only uses the function listed above without changing the object pseudo key. The other two is called randomized SCADDAR and bounded SCADDAR, which in different ways calculates new values for the object key.

The function for remapping X in scaling step *i* is for randomized SCADDAR

```
$X_i$={
    if ($(p\_r(X_{i-1}) \bmod N_i) < N_{i-1}$ )
        $p\_r(X_{i-1}) \times N + S_{i-1}$
    else
        $p\_r(X_{i-1}) \times N + (p\_r(X_{i-1}) \bmod N)$
    }
```

Where p_r is a function for generating pseudo-random numbers.

The function using bounded SCADDAR is

```
$X_i$={
    if ($(X_{i-1}/N_{i-1} \bmod N_i) < N_{i-1}$ )
        $X_{i-1}/N_{i-1} - (X_{i-1}/N_{i-1\,i} \bmod N_i) + (X_{i-1} \bmod N_{i-1})$
    else
        $X_{i-1}/N_{i-1}$
    }
```

Of the two functions listed above the randomized SCADDAR is considered the best. The bounded SCADDAR will only perform in a satisfying way when there are not too many nodes in the cluster

The main drawback of the SCADDAR algorithm is that it is required to keep track of all the generations of mapping functions. In some cases the algorithm will need to use all the functions to find the right node. In highly dynamic setting this can eventually become a large number of functions. It will also be a very ineffective lookup if all the servers are added one at a time.

The positive thing about the SCADDAR algorithm as opposed to the more famous linear hashing and extendible hashing is that during scaling it will guarantee that the cluster of nodes has a good load balance.

4.5.1. Converting to other types of data

The SCADDAR algorithm was originally developed for continuous media, like video and audio. The algorithm based the distribution on dividing the objects into blocks placed on different servers. In a search engine setting the documents are usually placed whole on servers. To modify the SCADDAR algorithm to this situation it might be possible to cluster documents into groups, where the groups have a unique id that can be used as a seed. The documents then get the role of blocks as the normal SCADDAR uses.

An alternative is to just use the scaling method of SCADDAR, and then use a normal hash function instead of using random number generators to get a pseudo key.

4.6. Hashing/Range-partitioning

This method is not taken from a published article but, was presented to me by one of my advisors. It combines the two methods most used to distribute documents over a cluster of computers, namely hashing and range-partitioning. The basic layout is that each node gets a share of the range of values the objects key can have. The documents is then stored at the node responsible of the key value to the document. In general the distribution of values for the keys is not uniformly distributed, and therefore a pure range partitioning scheme will get a skewed load balance among the nodes. To counter this, a hash function is used on the document key creating a pseudo key. An important property of hash functions is that they randomize values, and therefore spreads the values uniformly throughout the possible range of values.

Using a range partitioning as the basic structure also creates the possibility to change the range each node is responsible for. If a node gets a very large portion of the documents stored, the system can reduce the range this node is responsible for. This procedure can by done at all stages of a system lifetime. This property the algorithm has is an advantage over pure hashing methods for distributed systems. This is because balancing of document loads for these algorithms only happens when the system is scaling, and not when there is data skew.

With the use of range partitioning, there is need for a directory to determine the location of the documents. The directory in this case is just for data placement and not for queries. If there are frequent document updates and scaling of the system, a central directory might not be able to handle the load. A possible solution is making the updates to the master node "lazy". This means that when the border between the nodes change, the directory does not get message about this right away. Later when the master node makes an insertion or deletion to the a server and the document key is outside the node range, the node will then send the operation to the neighboring node, and further sends updated boundaries to the master.

4.7. xFS

This method of distribution is from the xFS file system (Anderson et al., 1995). The distribution of documents used here has some similarities to the method with hashing and range partitioning. The main difference is based upon the fact that xFS is a file system, and handles data a bit different then distributed system with object storage. The distribution is mainly based on a set of managers. Each manager has responsibility for a set of a set of documents. Which documents that is mapped to which manager is defined by hashing of the document id. Further, to be able to change the mapping from document to manager, more hash keys than managers are created. This way the system is able to move keys more freely from manager to manager. This is quite useful if one manager has too much load, or if one of the managers should crash. The architecture described here is so far similar to the previous description about hashing/range-partitioning. One difference is that instead of a central node keeping track of the managers, xFS globally distributes a map of managers over all nodes, so that a client or server can send a document directly to the right node.

When a document is mapped to a manager, this manager does not store the document itself. The manager is basically a directory. xFS is a file system, and therefore does not store single documents. File systems usually stores files divided into blocks, and this is also the case with xFS. xFS is based upon functionality from RAID and LFS(Log-based File System). Each file is striped into blocks divided onto nodes just as in RAID based system. To limit the partitioning of files too much the cluster is divided into stripe groups. The managers in xFS store the locations of inodes. These inodes tells which node the different blocks are stored on.

The method of distribution from xFS can be adapted to search engine settings, by letting the managers keep directories for exact document placement instead of file blocks.

As with the previous method, this algorithm makes it possible to rebalance the load in case of data skew without changing the cluster size. This is done on the manager level. One opportunity that method gives which lacks for pure hashing/range-partitioning, is that since the managers keeps directories, these can move documents freely between the storage nodes. This keeps a deterministic document placement on manager level, but lets the actual storage done freely.

5. Theoretical Analysis

This section will present an evaluation of the different distribution algorithms introduced in the previous chapter. The analysis will examine the methods and show what strength and weakness the different methods have

5.1. Criteria for evaluation

The main categories the analysis is going to focus on are

- *Load balancing*
- *Lookup efficiency*
- *Data volume transported*

These categories will be the main evaluation criteria, but for individual methods some other characteristics might be described also.

Data distribution
This criterion emphasizes how the documents are partitioned among the different node. The difference between the most heavily loaded node and the least loaded node will be measured. The difference will be illustrated by a histogram of the node document load.

Lookup efficiency
The study of lookup efficiency will analyze how many operations the algorithms need to be able to find a document, or find out where a document will be stored. This analysis will look upon both the number of operations, and if there are risks of false lookups by the algorithms.

Data volume transported
This criterion focuses mainly on the scaling of the system and how efficient this is done. For the algorithms it will be analyzed how much of the total data volume which needs to be iterated through, but does not need to be moved to another storage node. Another element of this category is how big the data volume actually being moved. The volume of data transported should be at minimum and preferably just transporting data to new nodes.

5.2. Notation for analysis

For the evaluation of the methods there will be used a standard notation and set of variables used for all methods. For some methods there are also some extra variables in use. These will be introduced under the relevant sections.

Standard variables

- $S_{j:}$ Server/Node number *j*.
- N_0 : Number of nodes initially in the cluster. Σ S_j
- N_i : Number of nodes after *i* scaling steps.
- D_j: Number of documents in node number *j*
- D_{tot}: The total number of documents in the system

  $$\sum_{0}^{N-1} Dj$$
- D_{avg}: Average document load on the nodes
- D_{new}: Number of documents belonging to the new node(s).
- $D_{trans:}$ Number of documents moved
- D_{total_read}: Total number of documents read. This is basically the ratio between D_{new} and the capacity of the nodes that document are read from.
- M: Lookup efficiency. This is either the number of messages needed to reach the correct node, or the complexity of the computations. Which type depend on the algorithm
- X: Document key

5.3. Linear Hashing (LH)*

In LH* the redistribution of data to new nodes is done by splitting of old nodes. Because of this, there will be no document movement between old nodes and the number of document that needs to be iterated through is minimal.

An example of the growth an LH* based cluster will now be presented to better illustrate the strength and weaknesses of LH*.

The variables for the analysis are:

N_0	4
N_1	6
N_2	8
$D_{1...4}$	100
n	Split pointer

The figures below illustrate two possible situations during scaling of the cluster.

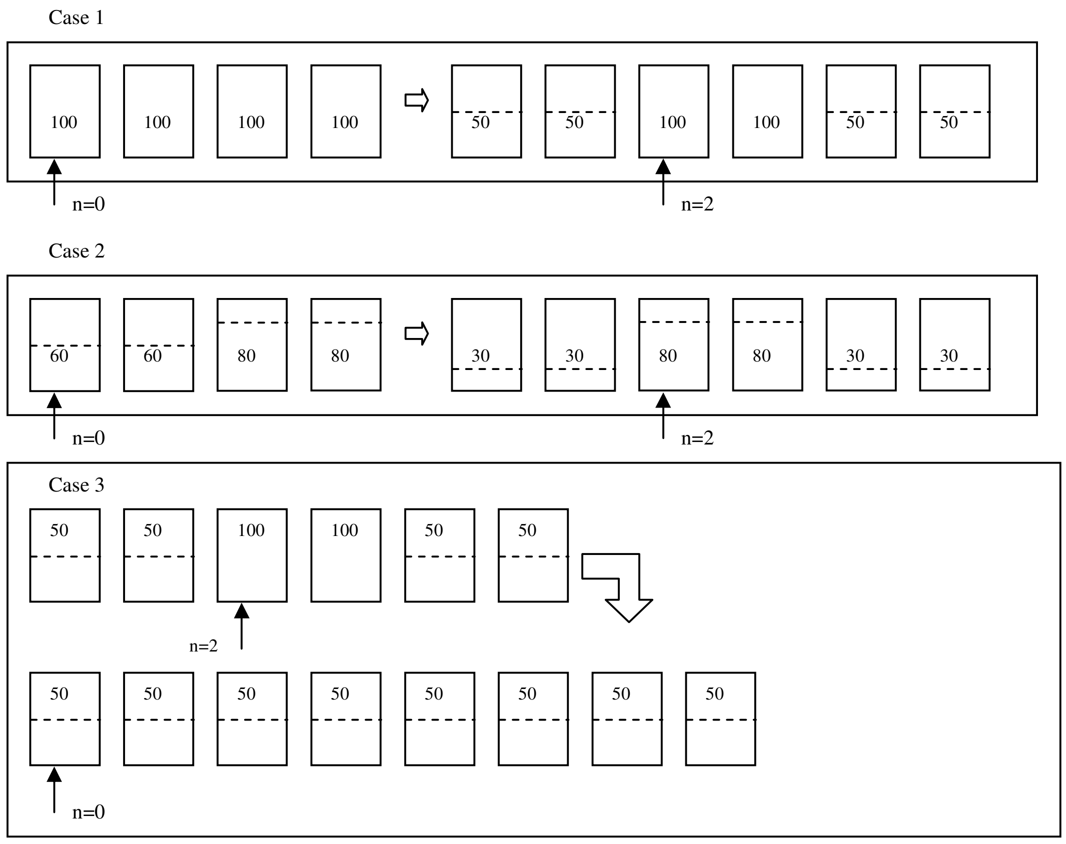


*Figure 5-1 3 different scaling situations with LH**

The first case shows an "average" situation where two nodes are split, thereby adding two new nodes. The example shows a clear drawback for the LH* algorithm. When the nodes have nearly equal load and a splitting is performed, the resulting overall load balance is that some nodes have roughly half the number of documents then others. In this case node number 3 and 4 has double of the load of nodes number 1, 2, 5 and 6. This situation will last until the number of nodes becomes $2^k * N_0$, where k is some arbitrary number.

The second example is to illustrate a "worst-case" for LH*. When scaling the system is initiated, the LH* algorithm has predefined which node that is going to split, and this is determined in a round robin fashion. This case shows where the node that the split pointer points to is relatively lightly loaded, while some are nearly full. Ideally the most heavily loaded should be the one that split. When the scaling and redistributing of the data has been completed, the result is a cluster with extremely bad load balance. If the load balance is not close to perfect before scaling, the splitting will then possibly make it worse.

Case 3 is a further expansion of the cluster in case 1. In this case the number of nodes has reached a number equal to the power of two multiplied with the number of original nodes. As the figure shows this leads to a possibility of a perfect data distribution.

Since there is no master server to keep track of the cluster level and split pointer for the clients and servers, all nodes have there own image of these variables. It will eventually come to situations where these images become outdated. The images belonging to the nodes are only changed when the nodes are split. The different nodes does not know which other nodes have the same values for the global variables. A node can therefore send a document to a node, believing this uses the same variables as its own. If the clients send documents to wrong servers, the system requires at most 2 internal messages to find the rights server.

The data volume transported during scaling with LH* is only the minimum number of documents. It is only the documents moving to the newest nodes that are being transported. That is, the ratio of documents moved is D_{new}/D_{tot}. The second element of the data volume criteria is the ratio of documents read that are actually moved. In LH* the documents that are moved to a new node, only comes from a single node. To find the correct documents, all the documents on the node needs to be traversed. The ration between documents read and documents moved are D_{new}/D_j, where j is the node number.

Analysis of LH*	
Data distribution	*Average:* Some nodes have half the load of others *Worst case*: Extreme data skew as in case 2 *Best case:* Perfect distribution when $N_i=2_i * N_0.$ Then $D_{avg}=D_{tot}/N$.
Lookup efficiency	Max 2 internal messages + request and answer. M= *4 messages*
Data volume transport	Number of documents moved $D_{trans}=D_{new}$ Ratio number of documents moved against documents read. $D_{total\_read}=D_{new}/D_j$

*Table 5-1 Analysis of LH**

5.4. Extendible Hashing (EH*)

In many aspects the performance of EH* is similar to that of LH*. In (Hilford et al., 1997) the LH* is also used as comparison when the EH* algorithm was defined. Just as LH* the EH* algorithm redistribute data to new nodes by splitting old ones. The difference between the two, is that while the splitting in LH* is done with a round robin strategy, independent of which node is full, EH* on the other hand split the nodes that are full. The cluster expansion by splitting gives EH* some of the same weaknesses as LH*. Since EH* does not split in round robin fashion, there is not some predefined situation where the load balance is optimal. There is still the problem that newly split nodes will most likely contain much fewer documents than the other nodes. Since nodes need to be full before they are split, there is not the possibility of extreme skew as for LH*.

To convert the original extendible hashing to a distributed environment, for EH* the central directory from the original algorithm is replaced by something called cache tables. The cache tables let each client or server perform a lookup for a document. The limitation of the cache tables is that they can become obsolete, since a cluster update does not update all the cache tables in the system. Because clients and servers can perform operations with obsolete information, there is a possibility that documents will not be mapped to the correct node right away. This is partly the same situation that occurs with LH*. The constant time lookup from the original algorithm need a few extra operations to find the right node. Each node in the cluster server or client stores some variables, which keep track of the minimum and maximum values for the cluster depth. For each lookup the nodes must compute the possible key values. This leads to some extra computation before a document can be sent to a node, but the amount very small. The maximum number of checks needed, is basically equivalent to traversing a binary search tree from root to leaf.

Using splitting when scaling, the system perform all reads from one node. When the node is split, all of the documents must be checked to find if they are staying on the node or to be moved. EH* has minimal data movement during scaling, and just documents belonging to new nodes are moved. Also the ratio between moved documents and unnecessary read documents are the ratio between buddy nodes document load.

Analysis of EH*	
Data distribution	*Average:* Some nodes have half the load of others *Best case:* Might get perfect distribution in some situations, but not some predefined situations: Avg node load : $D_{avg}=D_{tot}/N$.
Lookup efficiency	Worst case: Might compute all values from min to max for cluster depth. M=log N
Data volume transport	Number of documents moved $D_{trans}=D_{new}$ Ratio number of documents moved against documents read. $D_{total\_read}=D_{new}/D_j$

*Table 5-2 Analysis of EH**

5.5. $RUSH_R$

In the analysis of RUSH there are some additional variables in use. These are:

- r, the number replicas mapped to a cluster
- m_k, the weight of cluster number *k*
- n_k, the cumulative weight of all cluster added before cluster *k*
- w, the weight of a single server
- c, the number of clusters

The $RUSH_R$ algorithm uses a hypergeometric distribution to determine the number of documents mapped to a sub cluster. The function *H* used to calculate this uses the parameters $H(r, n, n_k + m_k w)$. The weight of the cluster is usually derived of the number of servers in the cluster and the capacity of the nodes. To make the hypergeometric selection deterministic, a pseudo-random number generator with the object id as seed, determine if a replica is to be mapped to the current cluster. Since the algorithm uses an iterative mapping procedure starting with the newest cluster, this clearly favors the newest cluster. The weight determines the probability that the current cluster in the iteration is the correct for a document. Figure 5-2 shows an example of a $RUSH_R$ cluster developing.

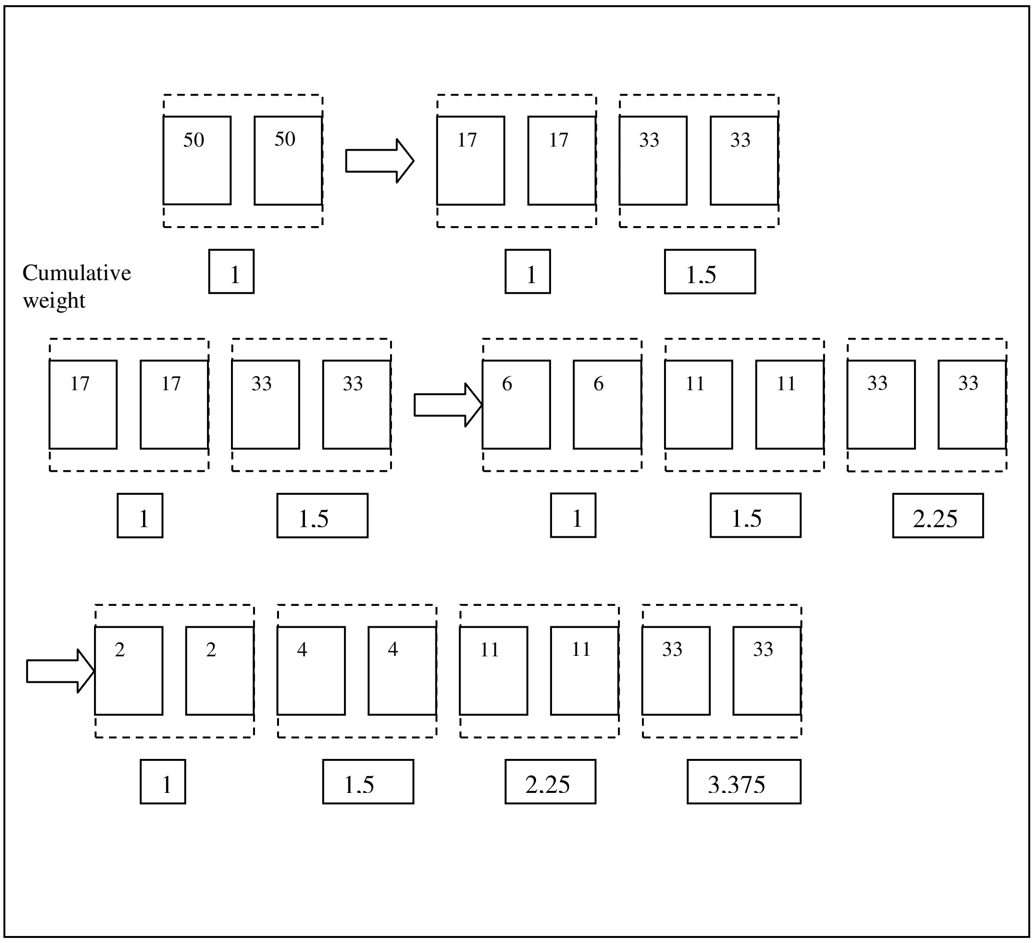


Figure 5-2 A cluster using RUSHR distributing documents across nodes.

The ideal load balance using the RUSH algorithm is somewhat different from other algorithms such as LH*. RUSH tries to place most documents on clusters with the highest weight, and not equal weight over every node. Internally in the sub-clusters the distributions uses random numbers and therefore gets an equal load balance. Average load balance pr cluster will thus be m_k/n_k+m_k

When the algorithm finds the correct node for a document, it uses a loop that checks every sub cluster in the system. This might be costly, and in worst case must check all the sub-clusters. On average, the algorithm will find the correct node much earlier in the iteration, since the newest sub-clusters have higher weight. The cost of mapping a document therefore depends highly on the sub-cluster weights. The worst case scenario is when the highest weight is given to the original cluster. In this case, most of the times the loop needs to iterate through all the sub clusters. When the algorithm has found the right sub-cluster, the remaining operation is not very costly. The main operations here are to initiate an array of integers with the size of the sub-cluster. Then further pick from that array pseudo-randomly the server numbers specified. The mapping of documents will in worst case take O(Rc+m) operations, where R is the replication factor for the documents. As mentioned earlier, the algorithm does not need to use all of the c iterations.

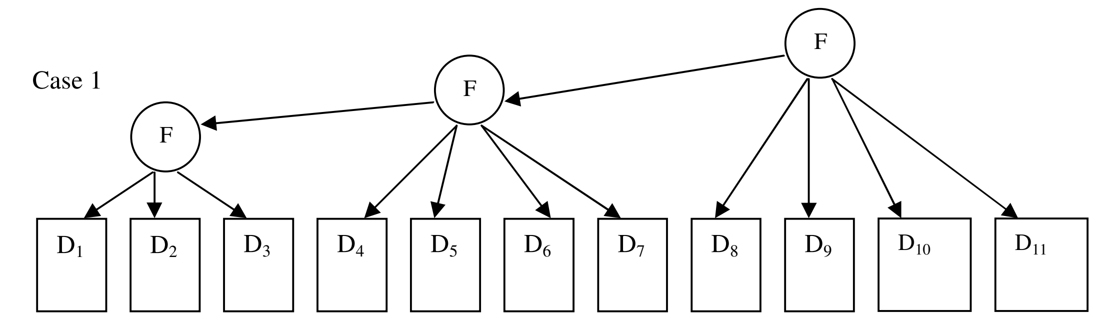
Analysis of $RUSH_R$	
Data distribution	*Average load:* $D_{avg}=m_k/n_k+m_k$ pr cluster
Lookup efficiency	*Worst case:* Iterations of all clusters M=Rc+m
Data volume transport	Number of documents moved $D_{trans}=D_{new}$ Ratio number of documents moved against documents read. $D_{total\_read}=D_{new}/D_{tot}$

Table 5-3Analysis of $RUSH_R$

5.6. SCADDAR

The SCADDAR algorithm uses a series of pseudo-random and remapping functions to determine the correct storage node for a document. A hash function will normally distribute data across all nodes equally limiting the data skew. The remapping function SCADDAR uses have a distribution of documents equal that of a complete rehashing. This therefore leads to a very well balanced load between the nodes in the cluster.

A big drawback with the SCADDAR algorithm is that it needs to keep record of the whole scaling history of the system, and use this when doing document search. In worst case, the algorithm needs to use the set of all hash functions to find the correct node. The method in which the nodes where added to the cluster determines the efficiency of a lookup. If the nodes are inserted one at a time, there will be as many functions as there are nodes. A more efficient structure would be to group nodes together, limiting the number of functions as much as possible.



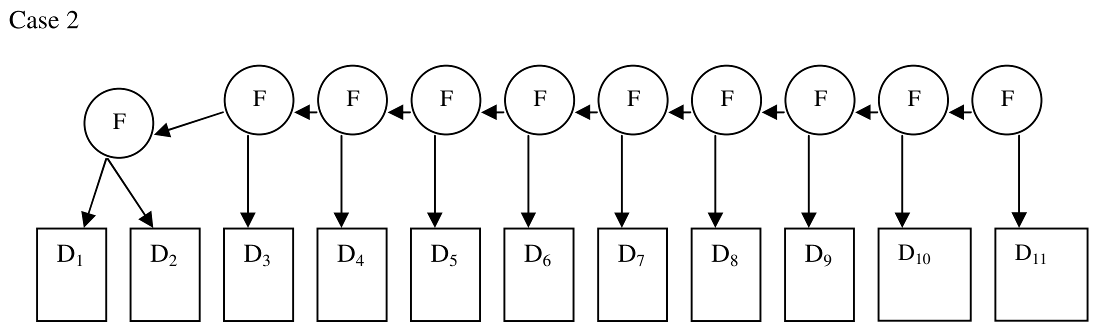


Figure 5-3 The structure of different SCADDAR clusters where the remapping functions F have different grouping of nodes.

The figure above shows two different situations that can occur when scaling a system with SCADDAR. In the first case the cluster has gotten a good layout with several nodes grouped together, and therefore the need of fewer remap functions. Case 2 is a worst case situation. Here the nodes where added one by one. This has led to one remap function pr node, and an insertion of a new document would possibly require traversing all the functions.

When the system is adding new servers and documents are to be moved, SCADDAR checks if the new mapping functions map a document to a new server or to an old server. If the mapping is to an old server, the document does not move and continue use the old mapping. Documents belonging to new servers change mapping functions and are moved to the new servers. This way the system gets a kind of rehashing, but with just moving a minimum number of documents. Then only D_{new} documents are moved. The SCADDAR algorithm might move documents from all of the old servers, and the algorithm therefore needs to read through all the documents.

Analysis of SCADDAR	
Data distribution	Distribution equal that of complete rehashing. Therefore equal load balance on all the nodes. Avg node load : $D_{avg}=D_{tot}/N$.
Lookup efficiency	Depends on the number of scaling operations. Worst case: M=N Needs to use a one function pr node lookup.
Data volume transport	Number of documents moved $D_{trans}=D_{new}$ Ratio number of documents moved against documents read $D_{total\_read}=D_{new}/D_{tot}$

Table 5-4 Analysis of SCADDAR.

5.7. Hashing/Range-Partitioning

With the combination of hashing and range-partitioning a system will start partitioning by hashing keys to a value range. This distributes the key equally across the possible values, which further get partitioned across the nodes. With an average data set, where the document keys are not tightly clustered to a few values, this method will give good distribution of data, because the hashing uses the whole set of possible values. There might be situations where dataset produce pseudo keys that are more tightly clustered. In this situation the use of equally partitioning of the key range will result in data skew, where some nodes get a large part of the documents. The range-partitioning gives a solution to this by changing the borders between the nodes. This will result in some data movement, but this is rather small.

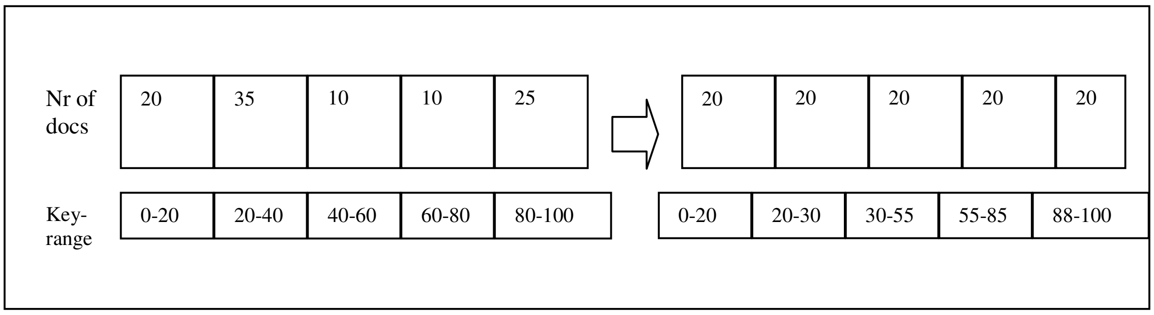


Figure 5-4 Cluster adjustment changing the range belonging to the various nodes.

This method was defined to use document lookup to the node without a central directory which keeps an updated view of the node borders. Every node has its own view of the borders on the cluster. The clients that send documents to the nodes also have their own image of the cluster. When the cluster is scaling up, the ranges belonging to the nodes will change. When this happens, the nodes involved will know of their change of borders, but this information will not be sent to the clients. This leads to cluster images for the clients becoming obsolete. When a client inserts documents to the cluster, it might send it to the node previously responsible for the key value in question. The receiving node will then send the document to the node which has taken over that key, and sends the updated information back to the client. This means that the maximal cost of document operation is 3 messages; one request, one images adjustment message, and one internal message in the cluster. This might not seem to be much extra cost, but it gets more costly if multiple borders are change that can leads to wrong data. If there are several clients, there is the possibility that all of them do the same wrong lookup.

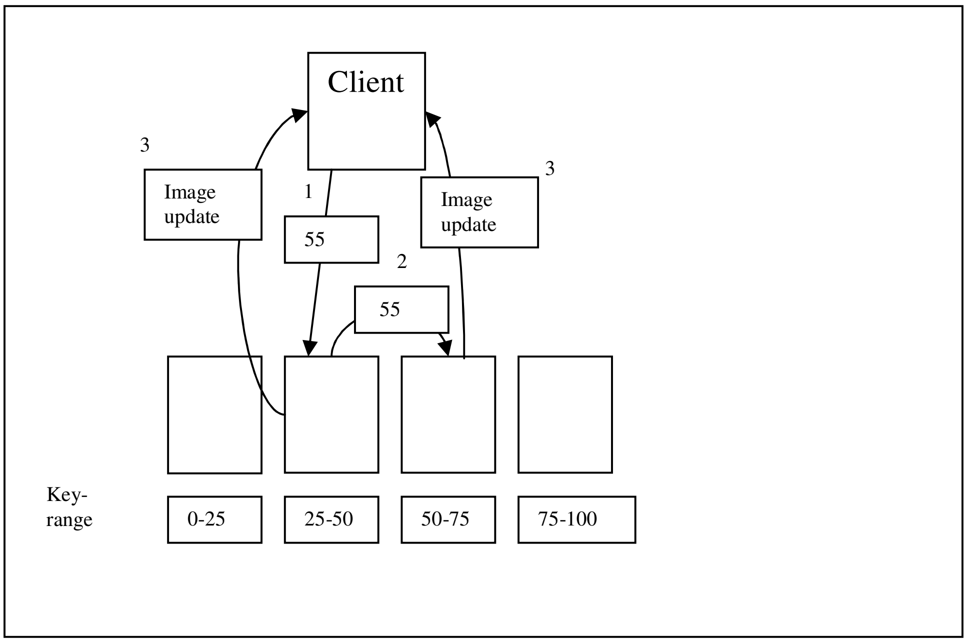


Figure 4-5 Messages required when a client uses an obsolete image of the cluster.

The organization of documents according to key range between the nodes can also be incorporated internal for the node. Since a scaling operation most often would require a change of borders, reading from nodes can be limited by organizing documents by key value. Using some data structure e.g. B-tree that enables range queries, the system needs only to read the documents that are to be moved; and not read through all the documents on the node.

For scaling operations, the hashing/range-partitioning method moves a large amount of documents compared to other algorithms. Unlike the previous analyzed algorithms this requires document movement between old nodes. The worst scenario is if the new node is inserted at the end of the range. The new node will be filled up with documents from the node previously placed at the end. When this is done, this node will in turn be refilled from other nods. This will keep on going until there is document migration from all the old nodes in the system. The part of the documents moved is

$$\frac{1}{N(N-1)}\sum_{i=1}^{N-1}(N-i)$$

This will be about half all the documents.

The cost of scaling can be reduced a bit by inserting new nodes in the middle of the cluster instead. With this method the new node is filled up by two different nodes. The total document movement for this method is

$$\frac{1}{N(N-1)}\sum_{i=1}^{N-1}(i/2)$$

Analysis of Hashing/Range-Partitioning	
Data distribution	$D_{avg}=D_{tot}/N$
Lookup efficiency	Max 2 messages to find the correct node + 1 image adjustment message M=3
Data volume transport	Number of documents moved $D_{trans}= \frac{1}{N(N-1)}\sum_{i=1}^{N-1}(i/2)$ Ratio number of documents moved against documents read. $D_{total\_read}=D_{new}$

Table 4-5 Analysis of Hashing/Range-Partitioning

5.8. xFS

This method derived from the xFS serverless file system uses an index with multiple levels. The top level with the managers works like a combination of hashing and range-partitioning. This give a deterministic placement of documents to the managers and the hashing gives a balanced load on the managers. The actual distribution to the storage nodes is done by directories distributed among the managers. The directories can place documents freely across nodes, and can therefore put documents where there is best capacity. This method also enables redistribution of documents without scaling the system. This means that the method can achieve a perfect load balance of D_{tot}/N pr node.

The lookup efficiency is fast. It requires maximum two messages to find or place a document. First it does a lookup to find the correct manager. Since the map of managers is globally distributed, this is done in one message. The next step is to send the message from the manager to the correct node.

With xFS it is the dictionaries that keeps track of document placement. During scaling, the most complicated process is to redistribute the responsibilities for documents from old managers to new ones. This is done in the same way as for other range partitioned indexes. It therefore requires

$$\frac{1}{N(N-1)}\sum_{i=1}^{N-1}(i/2)$$

of the elements from the managers to be moved. The difference from other similar methods is that here it is not the actual documents that is moved, but only elements in an index. When the manager map is updated, the managers can just move documents to new nodes freely. The number of moved documents is therefore at the minimum, D_{new}. The number of documents read unnecessary is also at the minimum. To keep the cluster load balanced, the moved document needs to be picked from every of the old nodes.

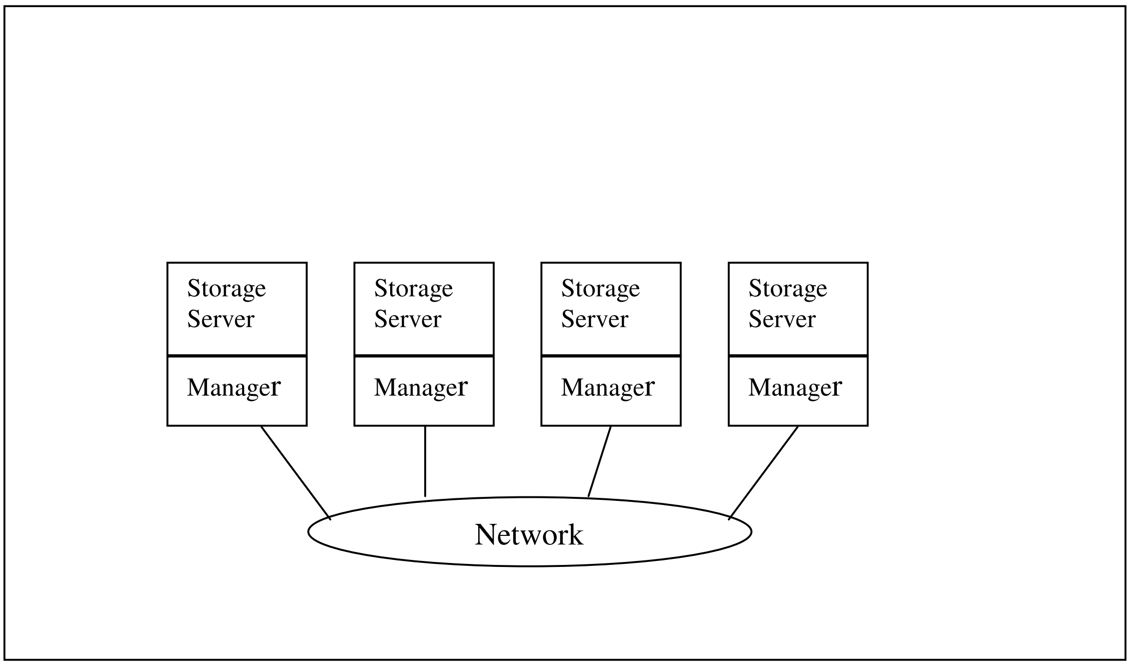


Figure 4-6 Cluster using xFS where each node is a storage server and a manager.

During lookup for documents, there is an average of 2 messages required. The mapping of document managers is globally distributed, to make sure clients will send documents to the correct node right away. When the manager has retrieved the document, it sends it to the final node responsible for the key in question.

The use of a dictionary, without predefined document placement, means that nodes can move documents of there own choosing. The only documents moved is therefore only those that are to be moved.

Analysis of xFS	
Data distribution	$D_{avg}=D_{tot}/N$
Lookup efficiency	Max 2 messages to find the right manager and one message from manager to storage node. M=3
Data volume transport	Dictionary without predefined document placement gives $D_{trans}=D_{new}$ documents moved. $D_{total\_read}=D_{new}$ documents read.

Table 4-6 Analysis of xFS.

5.9. Summary of Analysis

This section will now look at the different properties of the algorithms the analysis has shown.

The following table sums up the results from the analysis. These will be further commented in the following section.

	Data distribution	**Lookup efficiency**	**Data volume transport**
LH*	$D_{avg}=D_{tot}/N$ *	M= *4 messages*	$D_{trans}=D_{new}$ $D_{total\_read}=D_{new}/D_j$
EH*	$D_{avg}=D_{tot}/N$ *	M=log N	$D_{trans}=D_{new}$ $D_{total\_read}=D_{new}/D_j$
RUSH$_R$	$D_{avg}=m_k/n_k+m_k$ pr cluster	M=Rc+m **	$D_{trans}=D_{new}$ $D_{total\_read}=D_{new}/D_{tot}$
SCADDAR	$D_{avg}=D_{tot}/N$	M=N **	$D_{trans}=D_{new}$ $D_{total\_read}=D_{new}/D_{tot}$
Hashing/ Range-partitioning	$D_{avg}=D_{tot}/N$	M=3	$D_{trans}=\frac{1}{N(N-1)}\sum(i/2)$ $D_{trans}=D_{new}$
XFS	$D_{avg}=D_{tot}/N$	M=3	$D_{trans}=D_{new}$ $D_{trans}=D_{new}$

* *Best case*
** *Worst case*
Table 4-9 Summary of analysis

We will start with the LH* and EH* algorithms and sum them up together. This is because they share many probabilities both in function and in performance. LH* and EH* have the possibility of false lookups when searching for a document but, the performance cost of this false lookup is minimal since it in e.g. LH* uses maximum four messages to find the correct node. Also the computation required to find a document address is very short. On a modern day network of computers the sending of these few messages and the computation, is done very quickly. When dealing with movement of documents, these two algorithms can do it more effectively then some of the other algorithms. When a new node is added, this gets documents from just one old node. There is therefore less data that needs to be searched. The less processing needed for data movement compared to e.g. RUSH and SCADDAR comes at the cost of worse load balance. The analysis show the performance issue when using algorithms like LH* and EH*, which rely on node splitting when scaling a system. This leads to a very variable load balance dependent of the situation, document collection and hash function. If the hash function manages to partition document equally across the nodes, the algorithms will eventually get clusters with perfect load balance when the node count is a power of two. When the number of nodes does not satisfy this requirement, there will be some skew since some nodes will have approximately half the load of others. When the hash function does not manage to partition properly, there will be data skew in the cluster and as shown with LH* this can possibly get worse when the cluster is scaled.

For the RUSH_R algorithm the one potential performance degrading element is the document lookup. As opposed to the other algorithms in this thesis, the document placement is not completely deterministic, but also uses probability to partition documents. The algorithm will always start looking for a document in the newest sub-cluster. If a document is not found here, the search continues to the second newest sub-cluster. This process continues until the document is found, or it reaches the end of the sub-clusters. The algorithm does not actually search each sub-cluster but is determined by computation if a document exists. However but this computation must be done for each sub-cluster, which leads to potentially many computations.

A difference between the RUSH algorithms and the other algorithms studied in this thesis is that the ideal document partitioning is not equal load for all nodes. RUSH is design for heterogeneous cluster, and will try to have increased load on the newest nodes which most often have a higher capacity and performance than the old ones. When scaling RUSH moves documents from all existing node to the new ones. Because of this all document must be traversed to get the documents to move.

SCADDAR has a similar problem as RUSH. In the same way the process of placing or finding documents might require some computations. When there are many generations of hash-functions, the SCADDAR algorithm might need to use them all to find the location of a document. With SCADDAR the load balance among the nodes are kept equal, as long as the hash-function is good enough. There is not any limited situations that give perfect load balance like for LH* and EH* because documents are picked from every old node. This also leads to much work, since all documents must be searched to get the documents to be moved to new nodes.

The methods with Hashing/Range-partitioning and the partitioning algorithm from xFS is very much alike, and in many ways the same. The combination of hashing and range-partitioning gives a perfect load balance, because the ranges can be modified, and the use of local directories gives a maximum of 3 messages to find a document location. The difference from the other algorithms and the part that might degrade the performance is the work that is done during scaling. When the size of a cluster is changed the borders of the partitions must be changed to. Using hashing/range-partitioning, this is a process that leads to a lot of movement of documents, and most of the documents are moved between old nodes. The method from xFS is a bit different, and has a bit more effective approach. This algorithm does not hash and range-partition actual documents but only the responsibilities for them. The responsible nodes keep a directory of the relevant documents and place them where they want. This way a scaling of a cluster only updates the index, and the actually document locations can be gracefully changed by the managers which move only minimal number of documents.

6. Experiment

The goal of this experiment is to evaluate different methods for distributing data across computers nodes. We want to see if the results from the analytic evaluation are correct and also in what degree the pros and cons found in the analysis is transmittable to an actual system. Among the things we will look more into is whether performance is affected for LH* in terms of data skew and if the increased computation makes SCADDAR and $RUSH_R$ slower.

6.1. Experiment description

Different ways of doing this experiment is either to do a simulation of the methods, or to set up a cluster of nodes. In the simulation method all the operations and components will run on a single computer. The nodes will then be instances of a node class, which then will have documents allocated to it. The physical storage of documents is following not possible, and it will thus just be an allocation of responsibility for the documents. The second alternative is to create a small scale model of a search engine cluster.

In this assignment we chose the last option, because it gives a more realistic setting, since it has the same pros and cons as a real life system. It also gives the possibility to use a larger document collection in the experiment.

Using the procedure mentioned above, this experiment will incorporate various elements connected to document placement you will find in a search engine.

The experiment will primarily study the algorithms LH*, EH*, RUSH, SCADDAR and the combination of hashing/range-partitioning, and study how they perform when distributing a large number of documents to a cluster of nodes that gradually expands in size.

6.1.1. Input for the experiment

The input for the experiment will be documents from one of the TREC-collections. TREC (Text REtrieval Conference)[1] is a conference that focuses on research on large scale text retrieval. In that context TREC provides multiple data collections, called tracks, which enable researchers on information retrieval to use them as standard test sets. The collection that will be used for this experiment is the Terabyte Track, also called the GOV2-collection. This data set consists of approximately 25 millions web pages in the .gov domain that was collected during 2004.

[1] Text REtrieval Conference http://trec.nist.gov/tracks.html

6.1.2. Questions to answer

- Data distribution
    - At what degree does algorithms like LH* give worse load balance than RUSH and SCADDAR for systems with a not ideal number of nodes.
- Document lookup speed
    - At what degree does RUSH and SCADDAR perform worse than LH* on lookup procedures.
- Volume of data transport during scaling
    - How much data are being read during scaling, which is not going to be moved
    - How much data are being moved during scaling

6.1.3. How to answer the questions

- *Data distribution:* The result from the distribution of data can be found by counting the number of documents per node. The load balance can then be found by creating a graphical representation of the document distribution.

- *Throughput:* The best way to get a comparison of the algorithms is to register the time it takes for the algorithms to place a document at the correct position. To simplify the recording of the lookup speed, the time taken between scaling and calculating the average time pr document is used. This differs from the analytic evaluation, where the number of operation was counted. This is because the operations of counting operations would not give better answer in a practical approach like this experiment, than a theoretical analytical approach would give. This is examined by calculating the throughput between scaling turn. This will then show in what degree the increasing number of nodes will affect the speed of the document search.

- *Volume of data transport:* The amount of data being transported during scaling can be collected by letting each node counting the number of documents allocated to that node, and the number of documents sent to other nodes. One could then sum the number of documents sent, and compare this with the number of document for the new nodes.

  The volume of data transported is primary affected by the number of documents moved. In real system data volume is reduced by using compression. This will not be incorporated in the experiment, since it does not affect the algorithms whether the data is compressed or not.

6.1.4. Simplifications

This experiment includes multiple elements of a search engine. It is not a realistic task to create a fully working search engine just to test some aspects of its behavior. To solve this issue, only parts of a search engine will be implemented properly. This is manly the functionality incorporating document placement. Most of the other modules will be fake modules that only simulate the desired behavior.

6.1.4.1. Query load

One element, which will be left out, is the influence query load will have on the nodes. In the experiment only the load for the storage capacity will be of interest. The processing load on the nodes from queries would complicate the experiment and since it is outside the scope of this thesis, this will therefore be ignored.

6.1.4.2. Scaling tasks

The task of scaling a cluster includes both increasing the number of nodes and removing nodes. All the algorithms are designed to handle both. To simplify the experiment the, focus will be on scaling by growing of the cluster, and it will therefore not incorporate functions for decreased cluster size.

6.1.5. How handling simultaneous sending and receiving documents

This is a small issue that needs to be cleared, as it can potentially complicate migration of documents, and in worst case create an inconsistency in the indexes. The case here is that if a node has to both send and retrieve documents, the documents involved in these operations must be separated from each other. The reason for this is that if a node receives a set of documents and creates a new index prior to retrieving and sending away documents, these new documents and the belonging index will be scanned unnecessary. A way to avoid these complications is to retrieve an image of the index when the redistribution starts. When moving documents, only the documents from this image will be scanned. The documents added to the node will be incorporated into the real index, while this image is unaffected. Worth mentioning is also that this is not a problem while using most of the algorithms. With the exception of the distribution methods incorporating range-partitioning, all the other algorithms only move documents from old nodes to new nodes, and therefore this will not be any problem. But with range-partitioning there is a lot of document movement between old nodes, and therefore this issue must be taken into consideration.

6.1.6. How it is implemented

The experiment will be carried out on a small cluster of 6 computers provided by the Department of Computer and Information Science. Each computer will have programs running which have the various algorithms implemented. In that way they can find out if a document sent to that computer is the right storage node. The computers are tightly clustered, so that they are able to send documents to the right node. The kind of partitioning used, is what is called document partitioning. This means that the whole set of documents are split into subsets, and each node has responsibility for a subset. The alternate way of partitioning called term partitioning distributes node responsibility based on part of the index. This experiment does not focus much on the index organization, and as document partitioning is seen as most scalable, this method of partitioning is used. In addition to the computers in the cluster, there will be an external computer that sends documents to the cluster, monitors the system and in that way can activate new nodes and start the scaling process. On the external computer, a client program will be running that provides the cluster with documents. In addition, an external computer will also collect data from the nodes, like the load balance in the cluster, time use from the algorithms and number of documents sent between the nodes. This external computer will likely be the authors own computer in the students work area.

The experiment starts with the client on the external computer sending documents to the cluster, which starts with only one active node. When a document is sent to the system, a node will be chosen based on one of the algorithms. When a node receives a document, a hash function will be executed on the document identifier to get a pseudo-key. With the pseudo-key as input, one of the data distribution algorithms is executed to find which computer is the right storage node. This node will then send the document to the correct node, based on the information the current node has available. Whether the receiving node is correct, depends on which algorithm is used, as not all guaranties correct placement on the first try. The receiving node will then store the document, and update a counter for the number of documents on the node and the available space left. When the client starts sending documents, a timestamp is set. At the time of scaling the cluster, the client sets a new timestamp. Between these two occurrences the client counts the number of documents sent to the cluster can then be calculated. The average time the cluster has used to receive the documents can then be calculated. This will give a general view of the time the algorithms use on indexing.

Each node has a limit of how many documents it can store. When this limit is reached, it sends a message to the external computer of this situation. Dependent on the algorithm, this computer will then start the process of scaling the system. Some algorithms start scaling based on total cluster load, while others start scaling when one of the nodes is full. Then a new node is activated in the cluster. The nodes get a message to recalculate the storage placement for the documents stored in that node, and if necessary send the documents to the appropriate node. Which node that gets this type of message depends on the algorithm tested. For LH* and EH* this will only be necessary for the node which has to splits, but for others like RUSH$_R$ and SCADDAR this message is sent to all nodes.

During this process the nodes monitor how many documents it sends away and store this in a log. When the scaling is finished, the experiment continues and adds more documents. As before the scaling indexing time will be collected and later used to calculate indexing efficiency. The comparison of the times collected before and after scaling will give an indication of the effectiveness of the algorithms, and in what degree the algorithms are effected by data load and cluster size. When the run of the experiment is finished, the external computer collects the number of documents stored at each node, and by that gives a view of the load balance.

6.1.7. Software

The programs running on the nodes will have multiple components. One component listens to a port for communication with other nodes and clients. When a document is received through this stream, the metadata is checked if it is the correct node. If this is the case, the document is indexed and stored on the disk. If the node is the wrong receiver, a new component takes over the document. This component has socket connections with every other node in the cluster, and uses this to send the document to what the node believes to be the correct node. The third major component is the one that monitors the node. Among other things, it monitors the document capacity. When this is reached, it calculates the new storage node for every document currently receding on the node in question. Thereafter, it makes sure that they are sent to a new location using the sender component. The final components are some remote classes and methods that are used during communicating with the client and the program monitoring the scaling.

The software will be implemented in the Java programming language. This is done because this is the language the author has the most experience with. This gives the experiment two ways of letting the computers communicate. The first one is to program the interface with sockets, and the other one is Java Remote Method Invocation (RMI).

7. Results

This chapter will discus and summarize the results from the experiment. The focus will mainly be on the questions outlined in the previous chapter. We will here describe the results associated with these.

7.1. Experiment execution

To generate input the GOV2 collection from TREC was available. This collection is quite large, approximately 426GB and 25 million documents. This was more than the combined capacity of the whole cluster used for the experiment, so only a subset of this was used. From this collection between 500 000 to 750 000 documents, dependent of the algorithm used, were distributed to the servers in the cluster. This was found to be a dataset big enough to give useful results. Instead of giving the limit of capacity for the nodes in bytes, the limit was set in number of documents. This limit was set to 125 000. This was a document limit that was big enough to give useful results, and to run the experiment with this limit of capacity did not take to much time.

During the running of the experiment, the program was allowed to grow until all six nodes were utilized, and the scaling server was given a message that a new node was needed. This clause then terminated the run and collected data from the nodes.

All methods analyzed previously in this thesis have not been tested. Various methods have a lot of similarities, and the goal of this experiment was to look at some of the properties of the methods. Therefore the distribution method from xFS is not implemented in this experiment. This is because it shares most of its characteristics with the hashing/range-partitioning method.

7.2. Algorithm results

This section will sum up the data collected from the various algorithms, and further present some graphs and tables to help point out some of the characteristics of the performance of the algorithms.

7.2.1. LH*

When running the experiment with the LH* algorithm, the overall cluster load was set to 75%. As opposed to the other algorithms, which start scaling the cluster when one of the nodes are full, linear hashing based algorithms do not start scaling before the cluster has reached a fixed load set globally for the cluster. First after this limit has been reached and a node is full will the cluster begin scaling. As the histogram for the cluster load balance shows, two of the nodes, namely node 3 and 4 have more documents stored on them then the actual capacity. This is a result of using a scaling policy based on global cluster state instead of the state of single nodes. Therefore using LH* the nodes must have available some kind of overflow storage. For this experiment this was not a problem, since the document limit set was a lot smaller than the actual physical storage allowed.

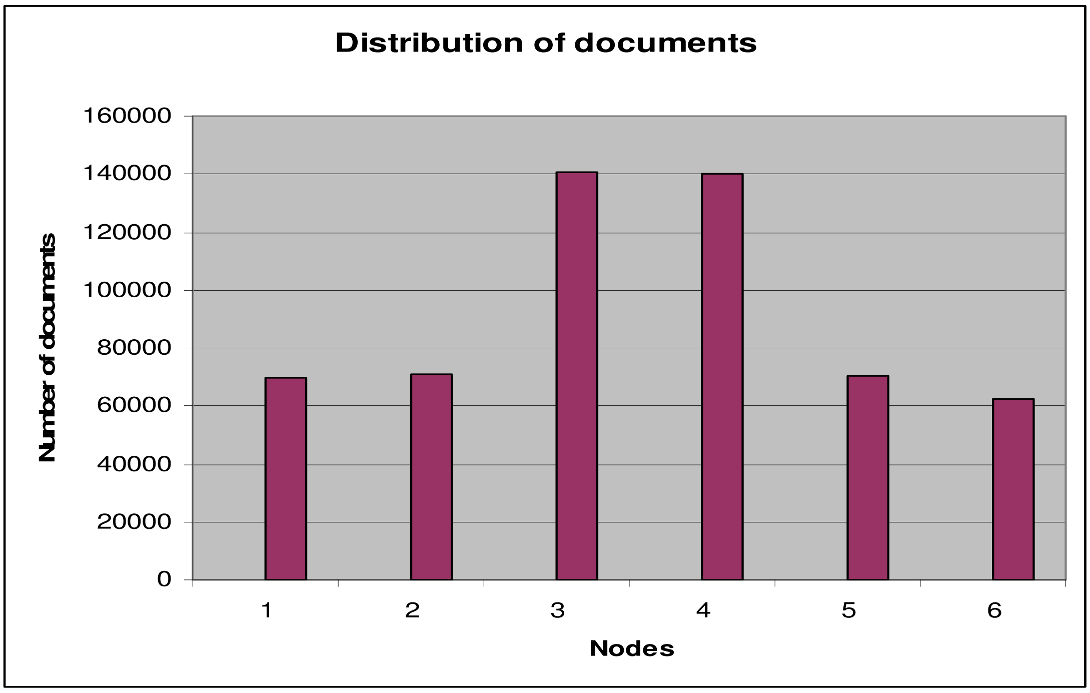


*Figure 7-1 Distribution of documents using LH**

This load balance from the experiment also shows that the distribution of documents across the cluster equals that of the theoretic analysis. Using six nodes the documents, growing of the cluster using splitting have led to the situation that node 3 and 4 has roughly doubled the load of the other nodes. This supports the theoretic analysis that perfect load balance will only be accomplished when the number of nodes equals the power of 2 of some number. Another element which can be observed from the graph is that because of the requirement that the cluster load needs to be past a certain limit before scaling, we can see that the number of documents on node 3 and 4 is past the actual node limit.

	Split no1	Split no 2	Split no 3	Split no 4	Split no 5
Document count before split	125000	124863	140688	124490	125003
Remaining docs after split	62240	62520	70485	61929	62698
% of documents moved	0,49792	0,500709	0,501002	0,497462	0,501572

Table 7-1 Percentage of documents moved pr node during scaling

The table above shows how many documents residing on a node before it is split, and the number of remaining documents after the split. This gives, as shown in the bottom row, the percentage of documents that are moved during scaling. Furthermore, it shows that the results from the actual implementation of the LH* algorithm follows the theoretical analysis, stating that roughly half of the documents residing on a node is transported to a new node. When looking at the top row, one can also observe that the splitting nodes does not necessary be one the nodes that become full first. On splitting number two and four, one can see that the nodes that were doing the splitting was not yet full. One can also see that for splitting three, the node in question was past it capacity.

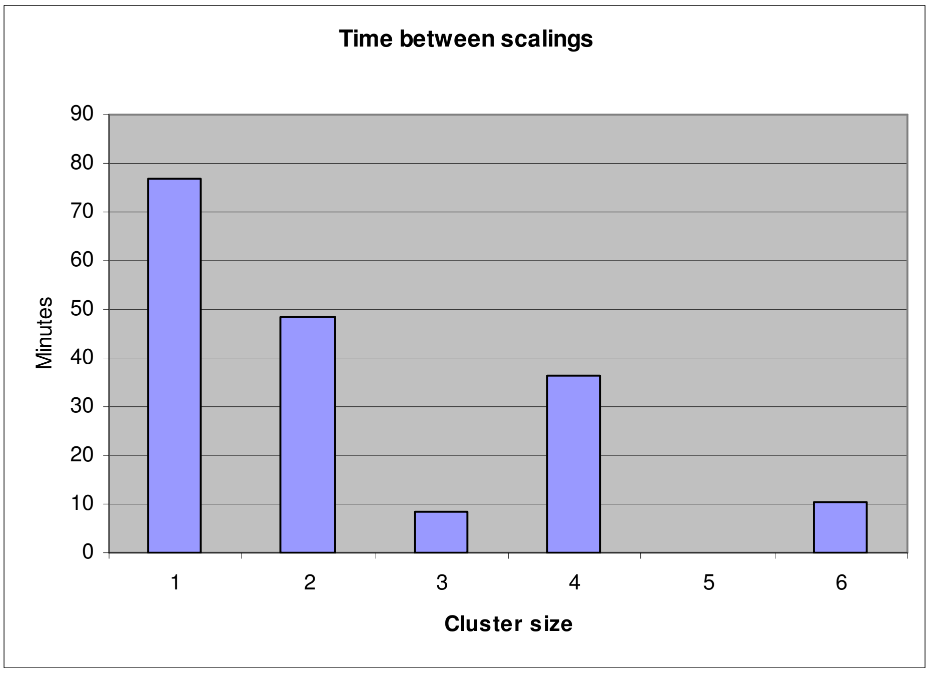


Figure 7-2 Time it takes between each time the cluster needs scaling.

The graph shown above shows the time in minutes used for indexing before the cluster needs an additional node. As can be seen on the graph, the times are rather variable. Especially period numbers 3, 5 and 6 are very small. The reason for this can be explained through the lifecycle of clusters using linear hashing-based algorithms. Before the second scaling, the cluster contains two nodes that have approximately equal document load.

They will therefore become full at the same time. When scaling number two occurs, the first node will split while the second node still is full as the indexing continues. The period between second and third scaling is then basically waiting until the cluster load reaches its required level, as the cluster already has a full node. This is also the case for scaling 6.

The graph shows that the fifth scaling starts instantaneous after the fourth one. This is the same reason as mentioned for the third scaling, but in this case the cluster already has reached its required load and therefore does not need to receive more documents.

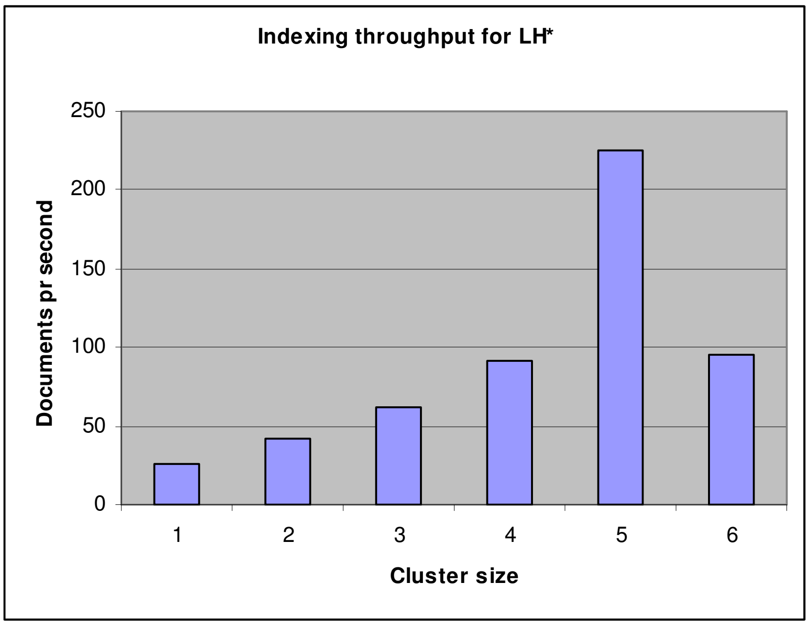


*Figure 7-3 Indexing throughput for LH**

When studying Figure 7-3 presented above, there seems to be a certain trend in the throughput that it is steadily increasing. That is, except for column 5. If we take notice on the stages of the cluster, from Figure 7-2, we see that the cluster has this size for very short period. With small data values like for this round, the addition of just a few documents would greatly affect the calculated throughput.

If we go back to the trend in throughput development, the steady growth can be explained that as the cluster growths, the job of indexing can be shared by more nodes. Since the throughput increases, this would indicate that the job of indexing document is constant. This would mean that the time it takes to calculate server numbers is independent of cluster size.

This graph also presents a limitation of this experiment; the cluster is not able to scale very far. Had the algorithms been tested on larger clusters, the outlier values like column 5 in this case would more clearly stand out as an outlying value.

7.2.2. EH*

As described earlier in this report, the EH* algorithm shares some characteristics with LH*. This is, among other things, the process of scaling the cluster through splitting. The result of this is apparent in the histogram of the load balances, as it is similar to that of LH*. We see that some of the nodes have a load which is roughly double the size of others. As opposed to LH*, the splitting has not been performed in a predefined sequence. Therefore, as the figure shows, the nodes in question having a greater document load are not positioned by each other, but could literally have been any position in the cluster.

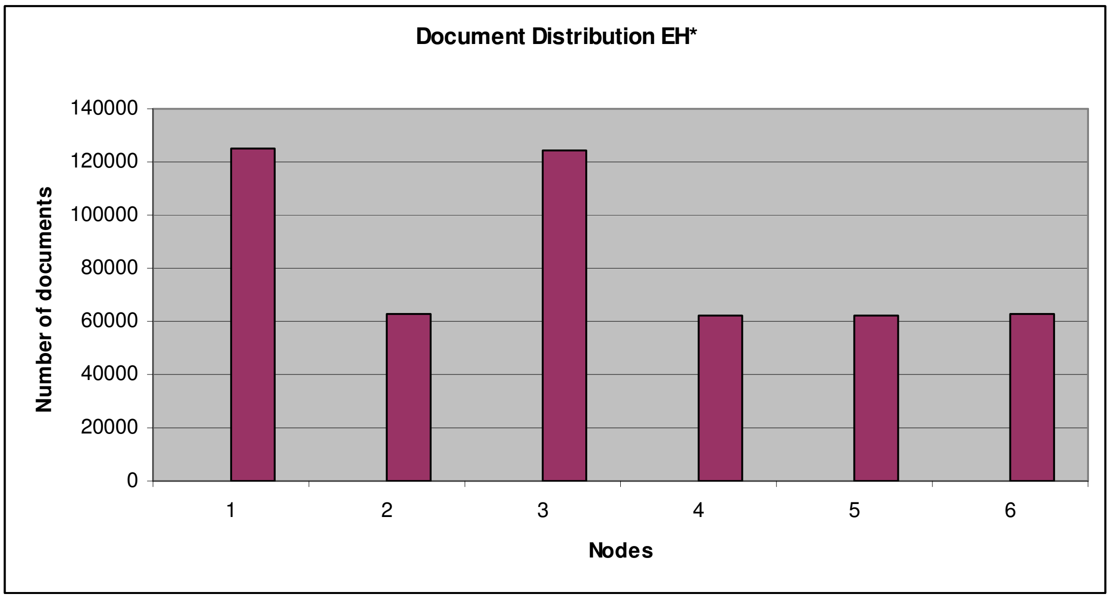


Figure 7-4 Distribution of documents across the cluster using EH.*

This kind of splitting, as Litwin in (Litwin, 1980) defines as premature splitting, gives some different behavior in the growing of the cluster as opposed to the more restricted and controlled splitting from LH*. When the documents are partitioned across the cluster to the nodes, the hash function does so that the load on the nodes grows approximately at the same pace. After a node is split, it will have the same load as the new node, and if the nodes receive new documents at approximately the same pace, this will lead to the situation where the nodes will become full at the same time. Therefore, after one node is full and has performed a split, the second node will do the same strait afterwards.

	Split no1	Split no 2	Split no 3	Split no 4	Split no 5
Remaining docs after split	62240	62622	62879	62593	62180
% of documents moved	0,49792	0,500976	0,503032	0,500744	0,49744

Table 7-2 Percentage of documents pr node that are moved.

The table above shows how large part of the documents stored on the splitting nodes which are moved during scaling. In the same way as the theoretical analysis, the table shows that the scaling with EH* transport roughly halves the documents resident on a splitting node. The document count before splitting is not included in this table, because for EH* it will always be the node capacity 125000.

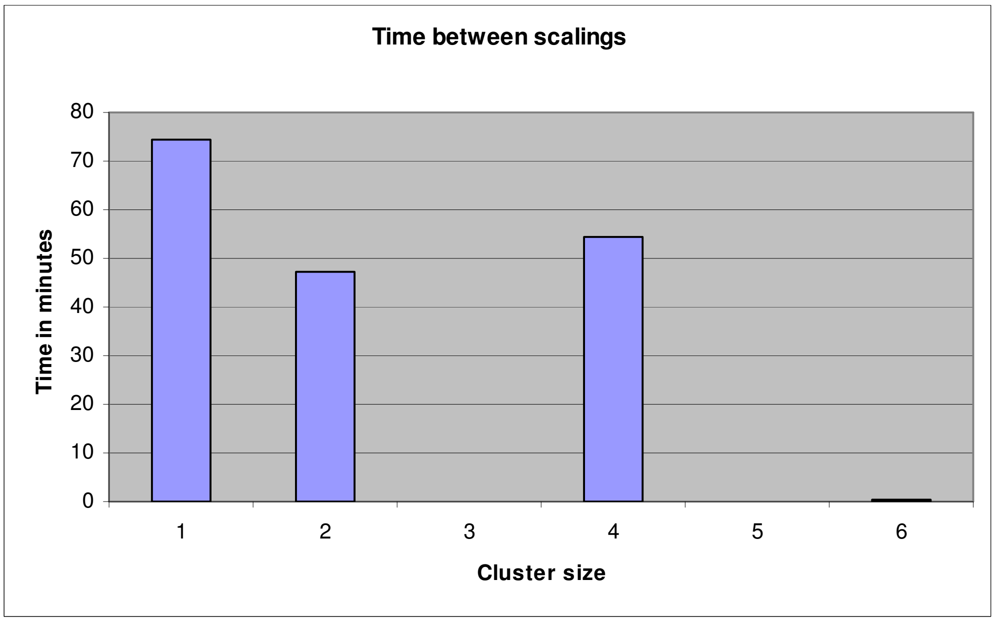


*Figure 7-5 Time between scaling rounds using EH**

When we look at the graph showing the time used between each scaling round, we see that EH* suffers from the same problem as LH*, and to a larger degree. One can see that half of the periods are about non-existing, meaning that scaling start immediately after the preceding scaling turn. As can be seen from the graph, the scaling start even faster after a preceding scaling then for LH*. The reason for this is the use of a minimum cluster load LH* uses, which limits to some degree the uncontrolled splits that can be observed here for EH*.

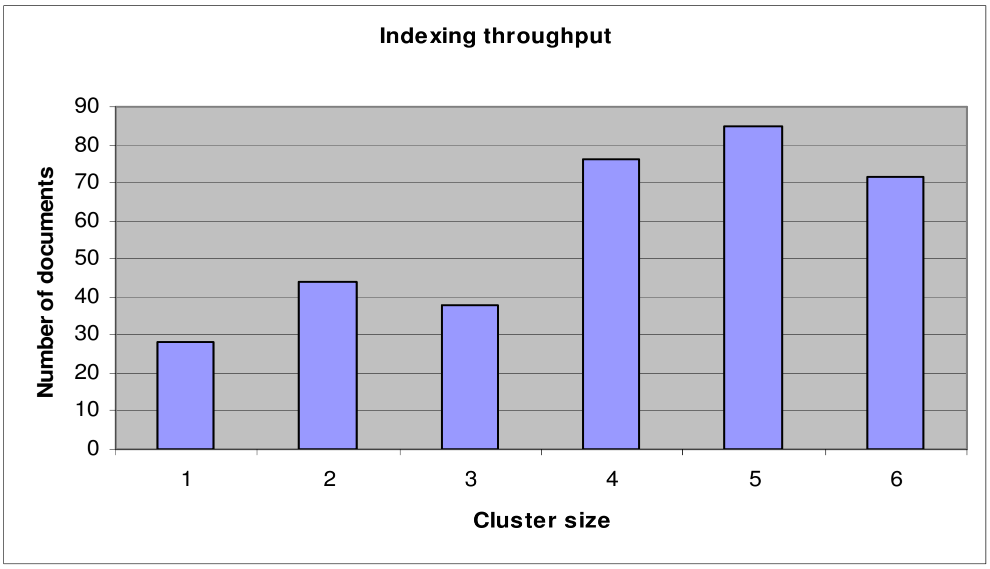


*Figure 7-6 Indexing throughput using EH**

When examining the throughput data for EH*, it must be observed equally as for LH*. At the times where scaling starts directly after each other the data collected would be affected by the short time span. It would still seem, that the there is a certain increase in the throughput. This would lead us to believe the time to index documents keep constant speed independent of cluster size, at the same time would the addition of nodes reduce the amount of work pr node.

7.2.3. RUSH$_R$

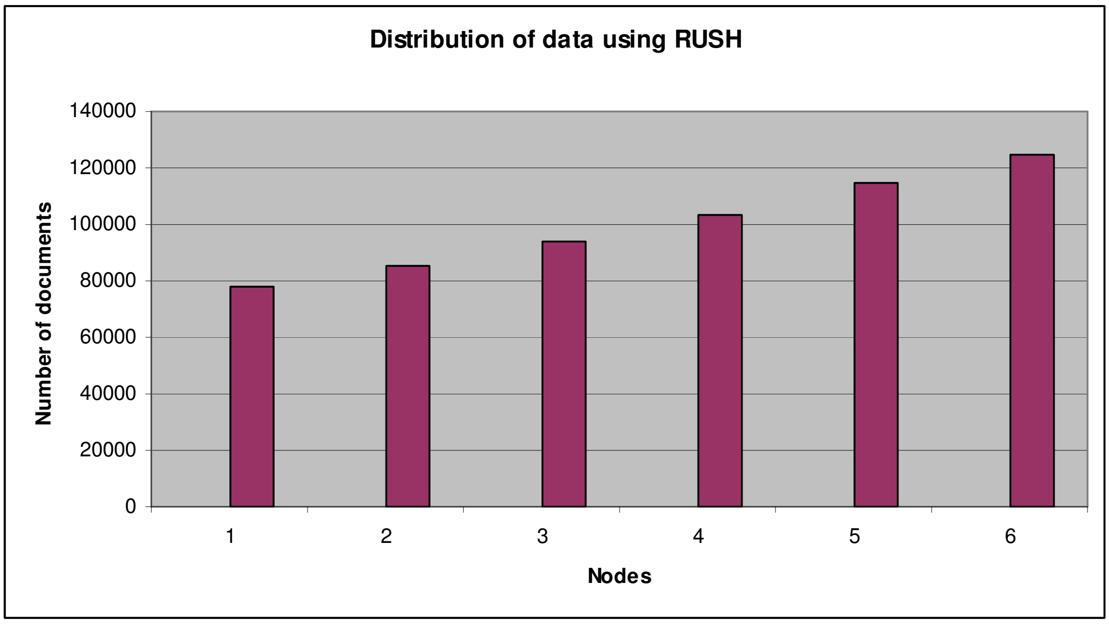


Figure 7-7 Distribution of data using RUSH$_R$.

The setup of $RUSH_R$ used here is a bit different from how the algorithm is designed. As described earlier, the total size of the cluster is just 6 nodes. This is a bit too small to implement with sub-clusters, as $RUSH_R$ is designed for larger systems. Therefore, each node is used as a cluster of its own. With the small size of the sub-cluster, there is less space to distribute the documents over, leading to a large number of documents on the new node. The distribution of documents using $RUSH_R$ favors newly added nodes, as can be seen on the graph. This weighting of nodes based on generations gives steady decrease in document load pr node as the nodes get older. During this experiment, each node is given an increase in weight of 1.1 times the previous one, so the difference in weight is not that big.

		Scaling 1	***Scaling 2***	***Scaling 3***	***Scaling 4***	***Scaling 5***
Node 1	Old doc Count	125000	113473	103275	93557	85742
	New doc Count	59466	71820	73542	71081	67796
	Percentage	0,524272	0,367074	0,287901	0,2402386	0,209302
Node 2	Old doc Count		125000	114000	103035	93901
	New doc Count		79719	81237	78288	74321
	Percentage		0,362248	0,287395	0,2401805	0,208517
Node 3	Old doc Count			125000	112938	103076
	New doc Count			88967	85781	81724
	Percentage			0,288264	0,2404594	0,207148
Node 4	Old doc Count				125000	114554
	New doc Count				95279	90960
	Percentage				0,237768	0,205964
Node 5	Old doc Count					125000
	New doc Count					98751
	Percentage					0,209992

Table 7-3 How much of the documents are moved during scaling for the nodes in the cluster.

The table above lists the data volume during scaling. This algorithm takes documents from all old nodes when the cluster is growing. The number of documents for the various nodes is smaller and smaller for each scaling process. The explanation can be that for each scaling, a node become less and less important. And as consequence of this, the percentage of the documents resident on a node that is to be moved, is decreasing for each round. This reduction of importance also means that it is only the most recent added node that becomes full, since this node always has more documents than the rest.

Total document transport					
	Scaling 1	***Scaling 2***	***Scaling 3***	***Scaling 4***	***Scaling 5***
Old doc Count	125000	238475	342276	434546	522273
Documents moved	65534	86936	98530	104104	108721
Percentage	0,524272	0,36455	0,287867	0,23957	0,208169

Table 7-4 The number of documents moved during scaling turns with $RUSH_R$.

During data movement at scaling, using the RUSH_R algorithm, it was defined earlier that the number of documents moved is close to ideal. That is, only documents transferred to new nodes need to move. When one compares the number of moved documents for $\text{RUSH}_{R,}$ as opposed to e.g. LH* and EH*, we see that more documents are moved for RUSH_R. The reason for this is that the family of RUSH algorithms sees new nodes as more important and therefore needs a bigger load. As the experiment shows, the transport volume for LH* and EH* is roughly half the capacity of a node. That is, if that node is full. The transport volume for RUSH_R is more that half since the new nodes must have heavier load than older nodes.

The figure above presents an overview over the documents transported during a scaling turn. The second column shows the number of documents moved to node number 6. When compared to the LH* and EH* algorithm, there are more transport over the network. The maximum number of documents with the mentioned algorithms is roughly 62500, i.e. half the node capacity. This is all transport between two nodes. As seen in Table 7-4, the difference becomes quite significant for the later scaling rounds. While the transport volume with RUSH_R is large, the overall load on the nodes is not that significant since the transport is shared among all the nodes in the cluster, but there is still the increased load on the network.

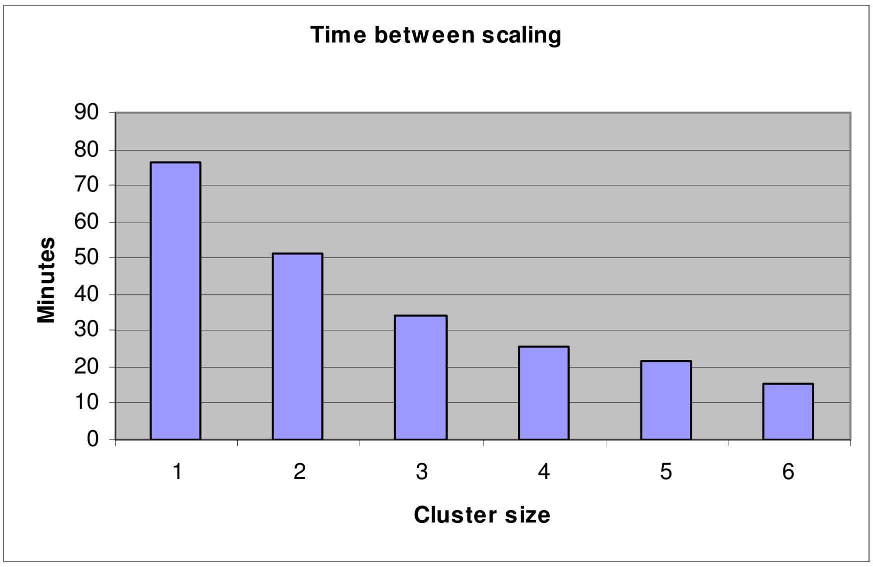


Figure 7-8 Minutes passing between each scaling with RUSH_R.

The graph above shows the time used indexing before each time the cluster grows. When comparing this graph with the graph of the previously listed algorithms, it shows that the development of the time passing is a bit more stable. . The reason for this difference is that RUSH_R gives a more balanced distribution of documents independent of how large the cluster is. This is, as described earlier, accomplished by taking documents from all old nodes to fill up a new node. The time it takes to fill up the nodes is clearly decreasing. When we see these results in connection with Table 7-4 presented above, we see the cause. Table 7-4, shows the overall document transport pr scaling. These numbers is actually the number of documents ending up in the newest node. As we see from the table, this number is increasing, meaning that it takes less and less documents to fill up the newest node and therefore to start a new scaling. Another case that can be taken into account is that as the cluster grows, there are more nodes to handle the indexing. If the client is able to provide a steady stream of documents to the cluster, the cluster as a whole will be able to process the documents faster.

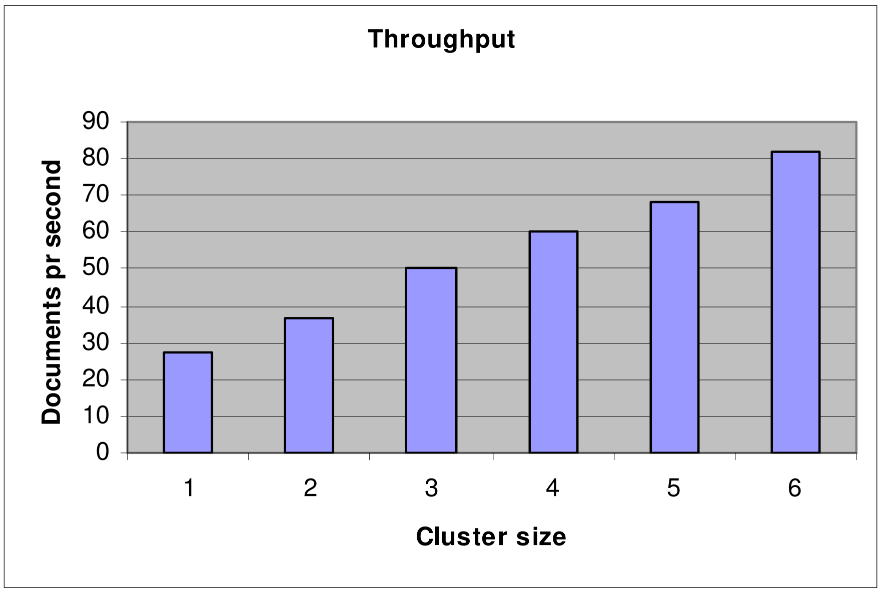


Figure 7-9 Indexing throughput using RUSH$_R$

When studying Figure 7-9, which shows the throughput using RUSH$_R$, one can see that the values correlate with the time passing between scaling turns. One can see that the relationship between them, that the throughput is inverse proportional with the time passing. This inclines that the number of documents added between each turn is more or less constant. This follows from the case when RUSH$_R$ redistributes documents; they are picked evenly from all nodes and thus keep the relative load between nodes constant.

7.2.4. SCADDAR

As mentioned in the chapter introducing SCADDAR, this algorithm is designed for continuous media objects which can be split and divided across multiple servers. For standard documents in a search engine, there is not an option to split the documents. Therefore to be able to test SCADDAR, the algorithm had to be used a bit differently. In the original SCADDAR, the functions are used to distribute blocks of the media objects and the keys are calculated using a random number generator. In this experiment, the key used was the hashed url of the document that was used as input for the SCADDAR algorithm to produce a new key. The SCADDAR functions were not designed for this use. It was found that the bounded SCADDAR was easier to use this way. Therefore, this function has been used in the experiment. Since the number of nodes used is so small, only six, this should still give credible results.

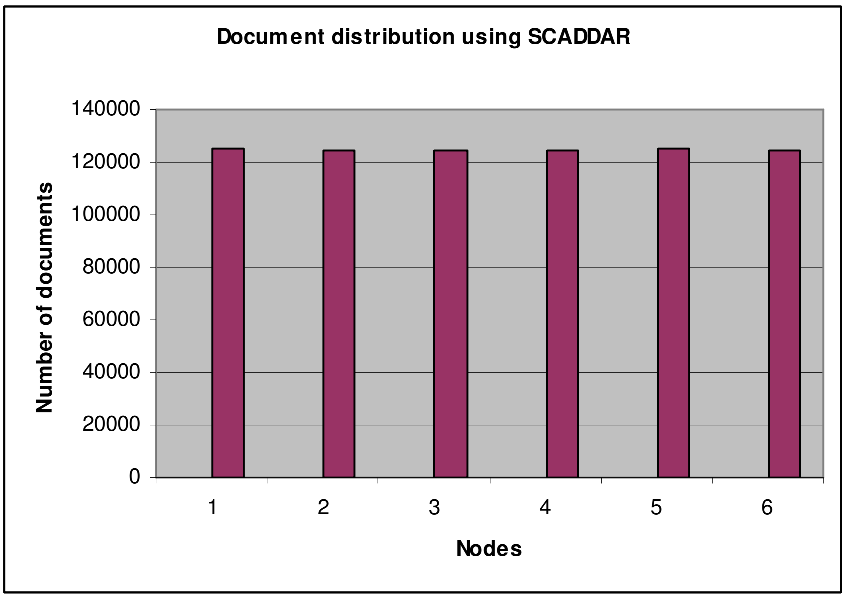


Figure 7-10 SCADDAR document distribution

Figure 7-10 shows the final document distribution after all six nodes are utilized. It clearly shows that SCADDAR manages to create near perfect distribution of data. Since all nodes are full, in addition to equally load, it means that the utilization of the nodes is perfect too. It is worth noting, that this utilization is in case of storage capacity and not processing capacity and handling of queries, as explained earlier in the report. Similar to hashing/range-partitioning, will the state of the cluster, using SCADDAR, always look like the in the graph in above. Assuming that the hashing functions in use give reasonable randomization, all the nodes in cluster using SCADDAR or hashing/range-partitioning will have approximately the same load all during their whole lifecycle.

.

Document transport pr node						
		Scaling 1	***Scaling 2***	***Scaling 3***	***Scaling 4***	***Scaling 5***
Node 1	Old doc count	125000	124844	124912	125000	125000
	New doc count	62240	83248	93874	100080	104187
	Percentage	0,50208	0,333184	0,248479	0,19936	0,166504
Node 2	Old doc count		125000	124306	124736	124126
	New doc count		83073	93312	99680	103709
	Old doc count		0,335416	0,249336	0,200872	0,164486
Node 3	Old doc count			125000	124799	124374
	New doc count			93694	99592	103669
	Percentage			0,250448	0,201981	0,166474
Node 4	Old doc count				124833	124398
	New doc count				99828	103548
	Percentage				0,200308	0,167607
Node 5	Old doc count					124877
	New do count					104082
	Percentage					0,166524

Table 7-5 Percentage of documents moved from each node during scaling using SCADDAR.

Table 7-5 presents the transport volume pr node for each scaling, and how large part of the total document load that is transported. As we can see, does the amount of documents that is removed from each node steadily decreases pr scaling round. Also when looking at the data from Table 7-6 shown below, one can observe that the total amount of documents moved is increasing. This can lead us to believe, that a further growing of the cluster would result in the document volume for new nodes will grow closer and closer to the node capacity. This growth seems to be logarithmic, so it would never actually reach this value. Part of the reason for this development is that only one node is added at the time, and seeing that less and less capacity is freed from old nodes with this kind of scaling, it would be recommended to increase the cluster by more nodes at a time. The justification for adding more nodes at each scaling is increased by the fact that small increases of cluster will lead to more computation during lookup, as was shown in the theoretical analysis.

Document transport in total					
	Scaling 1	***Scaling 2***	***Scaling 3***	***Scaling 4***	***Scaling 5***
Old doc count	125000	249844	374218	499368	622775
Documents transported	62760	83523	93338	100188	103580
Percentage	0,50208	0,334301	0,249421	0,20063	0,16632

Table 7-6 Document transport in total for whole cluster.

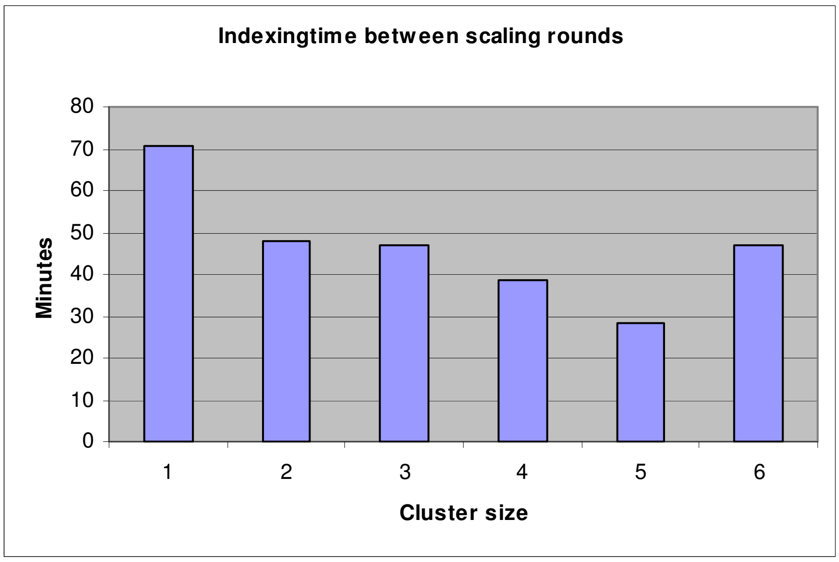


Figure 7-11 Time passing between scaling rounds

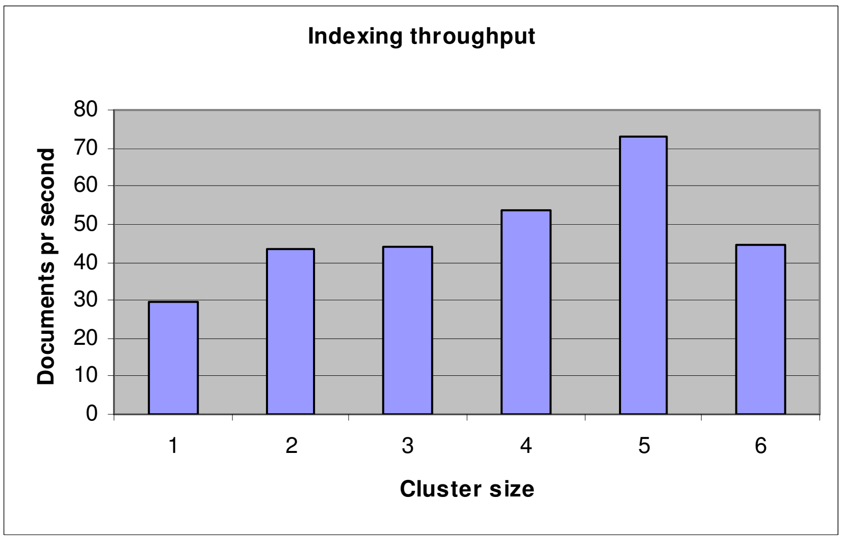


Figure 7-12 Indexing throughput for SCADDAR.

The Figures 7-11 and 7-12 indicate how the cluster size affects the throughput. We can see from the graph that until the last period, there is a decrease in time used and a following increase in throughput. The fact that the graphs follow the same development, but opposite of each other, shows that the time and throughput as measurements are inverse proportional. This proves that the number of documents indexed pr round is constant. The last indexing period stands a bit out from the rest of the graph. That there is a slight decrease in performance in this period can be traced back to the calculation of the node number. Bounded SCADDAR, used in this experiment, uses division in calculating object identifiers, and one of the inputs for this is the number of nodes. In this last period the number of nodes is six. Six can be factorized to two times three, meaning that documents mapped to nodes two and three for smaller cluster sizes will map to number six at the end. When SCADDAR calculates the document identifiers, the calculation starts at the smallest size and continues until the actual cluster size. Therefore, with cluster size six, more of the documents are mapped several times, which was not the case for earlier periods.

7.2.5. Hashing/Range-partitioning

When running this experiment with hashing/range-partitioning, the organization of the nodes during the run has been somewhat different than for the other algorithms. For the rest of the algorithms, when a new node is inserted, the new node is always inserted last in the list of nodes. In the theoretical analysis earlier in this report, it was shown that with range-partitioning the usual organization of nodes were not optimal. When inserting new nodes in the middle of the cluster, there will be fewer documents to move during scaling than if the nodes are inserted at the end. For this reason, the numbered nodes will be organizes as follows.

1 4 6 5 3 2

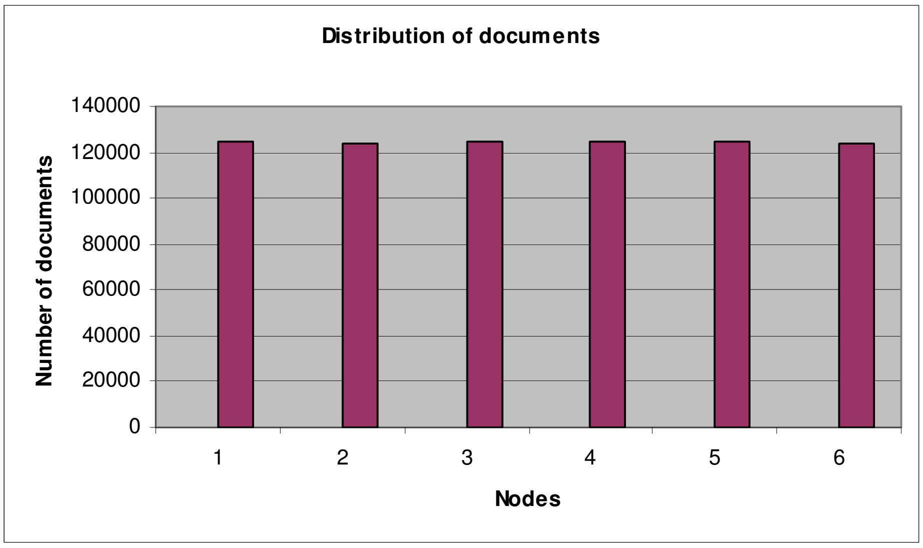


Figure 7-13 Distribution of documents using Hashing/Range-partitioning.

The distribution of data using range-partitioning is very dependent of the spreading of the data across the data range. As can be seen on the graph above, the hash-function used in this experiment managed this well, since the distribution of data when all six nodes have been utilized is near perfect.

Document transport per node						
		Scaling 1	***Scaling 2***	***Scaling 3***	***Scaling 4***	***Scaling 5***
Node1	Old doc. count	125000	124723	124635	124334	124006
	New doc. Count	62341	82969	93313	99621	103429
	Percentage	0,501272	0,334774	0,25131	0,198763	0,165936
Node 2	Old doc. count		125000	124239	124141	123555
	New doc count		83024	92904	99274	103146
	Percentage		0,335808	0,252215	0,200313	0,165181
Node 3	Old doc count			125000	125000	125000
	New doc count			62516	75471	83028
	Percentage			0,499872	0,396232	0,335776
Node 4	Old doc count				148645	143798
	New doc count				99874	103506
	Percentage				0,328104	0,280199
Node 5	Old doc count					123675
	New doc count					61800
	Percentage					0,500303

Table 7-7 Percentage of documents moved from each node during scaling.

Table 7-7 gives an overview of how large part of the documents residing on a node which will be moved to another node during scaling of the cluster. As the table shows, the percentages are rather variable per node and per scaling. This variation is a result of varying node positions in the cluster and the number of nodes in the cluster. If one looks at node number one, it starts with a total of 50% of the documents moved. This is consistent with splitting of the documents and an increase from one to two nodes. For this node, the percentage then decreases as the cluster grows until it reaches 16,5 % at scaling five. When comparing these numbers with the corresponding numbers for node two, one sees that they are similar. The reason for this is the development of the cluster. When new nodes are added, they are placed in the middle of the cluster. The two first nodes will then always have the position at the ends and therefore follow the same development. Studying the numbers for nodes three and four, one sees that the data corresponds to some degree there as well. If the cluster had grown even further, one would have observed a similar development for other node pairs.

Total doc transport by scaling					
Node	***Scaling 1***	***Scaling 2***	***Scaling 3***	***Scaling 4***	***Scaling 5***
1	62659	41754	31322	24713	20577
2		41976	31335	24867	20409
3			62484	49529	41972
4				48771	40292
5					61875
Total	*62659*	*83730*	*125141*	*147880*	*185125*

Table 7-8 Number of documents moved during each scaling.

When examining the actual number of documents moved and not just the percentage moved, one can see that it is rather different numbers from the other methods studied. As pointed out in theoretical analysis, will the hashing/range-partitioning algorithm move documents between old nodes in addition to filling up new nodes. This will lead to a large number of documents moved between nodes, as shown in the table above. When comparing these numbers with the algorithm presented earlier, one can see that hashing/range-partitioning provides large numbers. While the LH* and EH* algorithms having a consistent transport volume during all scaling processes only moves about 62000 documents, and RUSH transports approximately 108 000 at the most, using hashing/range-partitioning moves 185 000. This is a very large difference with LH* and EH* transporting one third of hashing/range-partitioning for the last scaling turn.

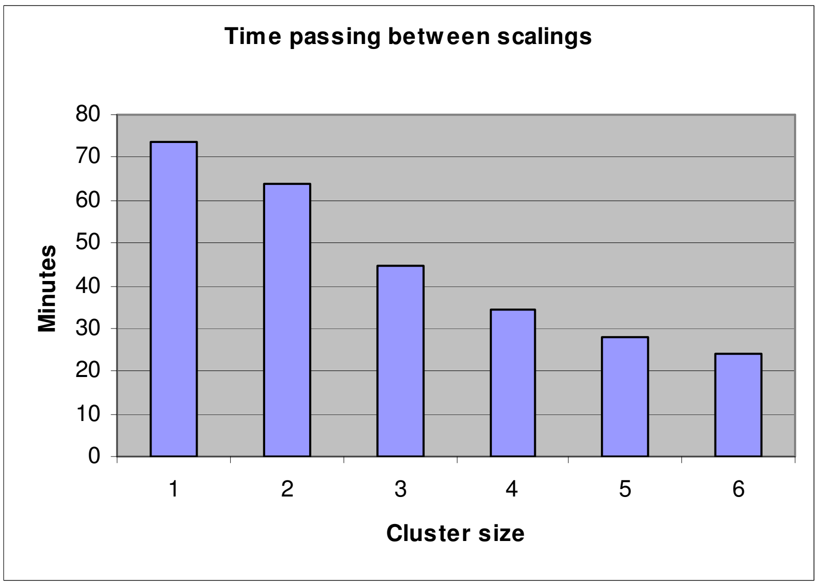


Figure 7-14 Minutes passing between each scaling with Hashing/Range-partitioning

The figure above shows the time it takes between each time the cluster scales. As can be observed, is the time it takes for the cluster to fill up decreasing. Using hashing/range-partitioning, all of the nodes will by more or less filled up at each scaling. This means that the number of documents added between each scaling is equal the capacity *c* of one node, following that one node is added each time. Granted that the time a node uses to index a single document is about constant, the time it takes to index the set of documents is then shared between the nodes. Furthermore, since the number of nodes that shared the job increases while the work required stays constant, this will lead to a development equal the graph shown above. The time development from the graph will also explain an increase in indexing throughput as show in the following figure.

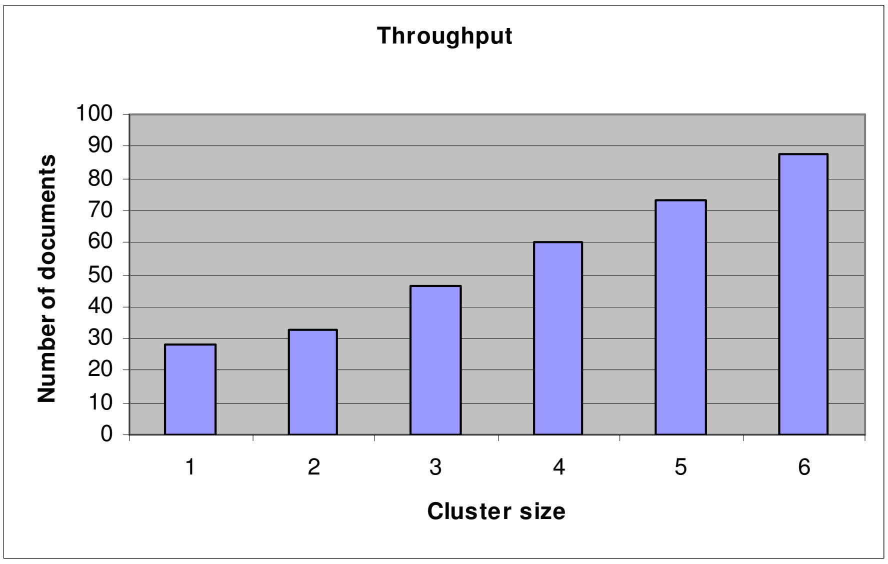


Figure 7-15 Indexing throughput using Hashing/Range-partitioning.

The development in throughput is, as stated above, a consequence of a constant number of documents added in decreasing time. This gives an increase in throughput inversely proportional to the time passing between scaling rounds.

7.3. Comparing the algorithms

In this section a comparison of the data from the methods will be presented with focus on document distribution, time efficiency and data movement, as described earlier in this report.

If we start looking at the results for the document distribution, it shows that overall there is very little difference in what the methods managed in practice in comparison to the theoretical results. As expected, it showed that the two first methods that rely on scaling by splitting would get a less balanced cluster, also was dependent on the number of nodes. Seen at the results from $RUSH_R$ it differs from the other methods, because it did not try for an equal document distribution, but the load is dependent of node age. The experiment showed that redistribution using all the nodes gave a more graceful distribution, were the difference in load did not came as extreme as for LH* and EH*. The last two methods, hashing/range-partitioning and SCADDAR, also tried for equal document distribution and based the redistribution on picking documents from all nodes. This gave at the end a perfect distribution of documents for the whole cluster.

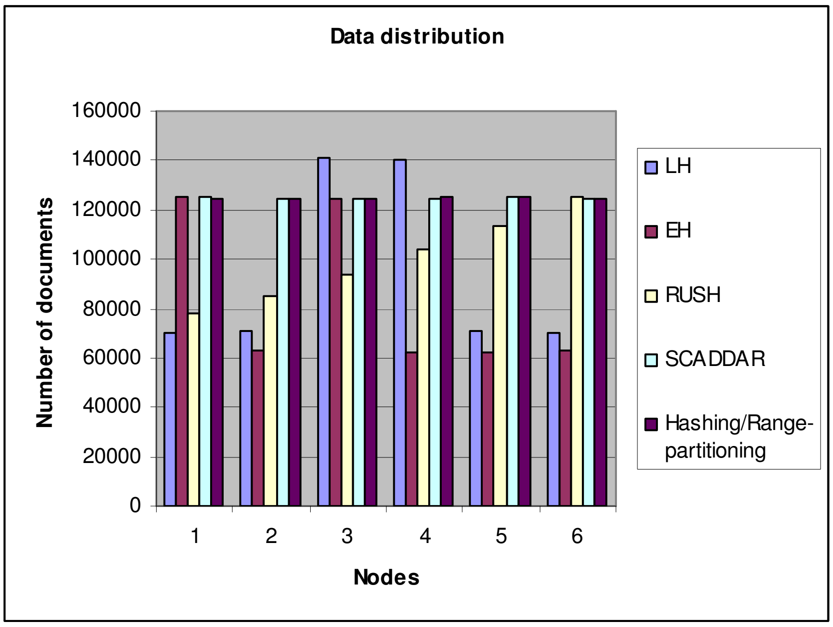


Figure 7-16 Comparison of data distribution

An effect of the difference in load balance is storage utilization. For the methods that keep perfect load balance during the whole life cycle, like SCADDAR and hashing/range-partitioning, the storage space will always be 100% utilized before scaling. When seeing at the methods that use scaling by splitting keeps the load balance is approximately between 66 and 75 percent. Here LH* lies on a balance of 75 % for each scaling, because of the lower limit criteria of cluster utilization build into the algorithm. The $RUSH_R$ algorithm lays on a utilization of about 80 %, placing it a bit better that LH*.

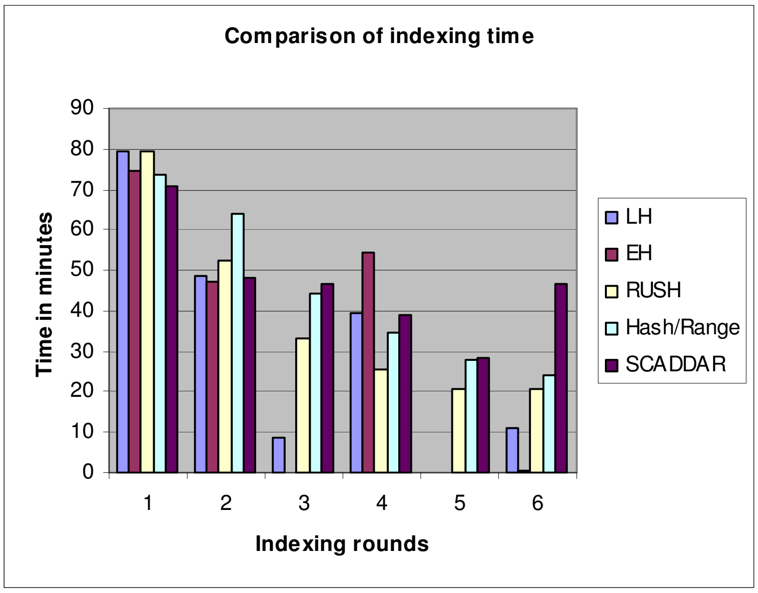


Figure 7-17 Summary of indexing time

Comparing the time used for indexing is a bit problematic, since the LH* and EH* methods give some unrealistic values in the cases where two rounds of scaling happen right after each other. For large scale cluster of 50-100 nodes situations like these would give an overall normalization of values, so that outlying values would have less effect. There are still some trends that can be pointed out based on the graphs. For the three methods which do not use splitting, it would seem that the $RUSH_R$ algorithm uses a bit less time between each scaling than the other methods. This can be explained by the fact that $RUSH_R$ gives different amount of documents pr node. The newest node will therefore fill up faster than with equal weighting.

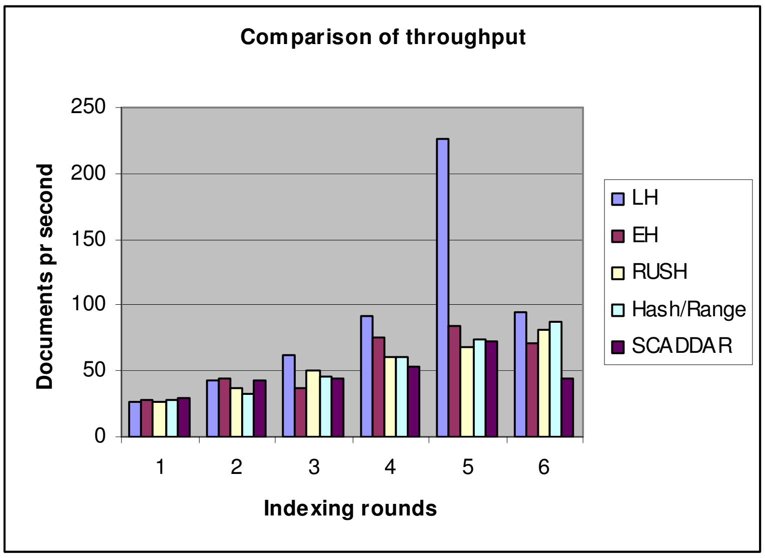


Figure 7-18 Summary of throughput

Studying the throughput, we see that there are some outlying values here also regarding the indexing time. However, this is of less degree, and it is therefore easier to pick out some differences. First of all, if we do not count indexing period 5, we see that LH* seem to have a little bit better throughput that the other algorithm. This might be explained by the fact that this algorithm does not have any complicated computing like $RUSH_R$ and SCADDAR, or some lookup in a directory like EH* and hashing/range-partitioning. EH* has for most of the times also a bit better performance than the other ones. For answering the question regarding $RUSH_R$ and SCADDAR stated before the experiment execution, it would seem that for a cluster of this size the increased computation has some effect on efficiency, but this is rather small.

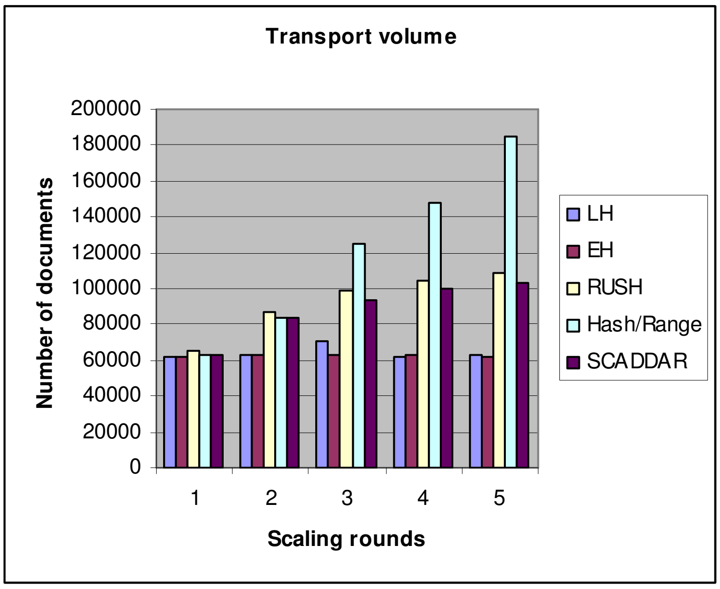


Figure 7-19 Summary of transport volume

Figure 7-19 shows a comparison of the document volume transported during scaling for each of the methods. It shows a clear difference of the effectiveness. The two methods that bases scaling on splitting of nodes show clearly the best results here. These methods keep the transport volume on approximately half the node capacity, independent of cluster size. The next best algorithms are $RUSH_R$ and SCADDAR. From the second scaling and further, there is a slow increase in transport volume. Also SCADDAR lays a few documents under $RUSH_R$. The last algorithm hashing/range-partitioning shows terrible performance here with a steep increase in transport volume as the cluster grows. This comes from the fact that this method is the only one to move documents between old nodes, in addition to filling up only the new one, as is the case for the other algorithm.

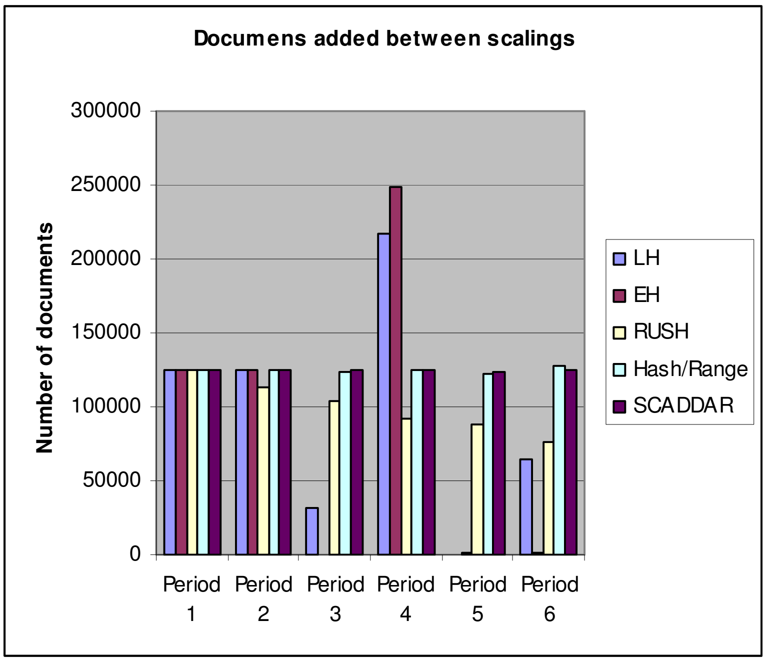


Figure 7-20 Documents added between scaling

Figure 7-20 shows how many documents added for each period in the cluster lifecycle. Taking a look at LH* and EH*, we see that the number of documents added are quite variable. For period three, five and six there are very few documents added. This is because when these periods start, there are already full nodes in the cluster waiting for the opportunity to split, as explained earlier for the indexing time. When reaching situations like this, it would be natural to add nodes in batches, so that nodes that scale right after each other get to split simultaneously. By doing this, the time used for scaling could be deceased. We also see that the documents added slowly are decreasing for the $RUSH_R$ algorithm caused by the increased number of documents for newly added nodes. For hashing/range-partitioning, there are a stable number of documents added. These methods share the same load balance for the cluster, keeping a near perfect distribution all the time. Because of this, at the time when one node reaches its capacity the, other nodes are not far behind. This means that for each indexing period, the documents added are always the number needed to fill up the new node.

8. Conclusion

We will now draw a conclusion of the applicability of the methods studied in this thesis. Using three different scenarios we shall try to determine how well the algorithms would handle these situations.

The scenarios are as follows

- News agency (Reuters, NTB) article collection: What characterize the workload for this use case is that it is a continuous addition of documents and frequently updating the newest articles. There is also a high frequency in document search for new documents.
- Wikipedia: For Wikipedia the frequency of document addition is similar to the previous scenario. In updating documents there is the same pattern with frequent updates for new documents.
- Indexing blog-posts: The workload for this scenario is similar to that of Wikipedia, but here it happens periodically.

What characterize all three methods is that there is high frequency in updating and adding documents. Often updates means that it is essential that the methods provide fast access. From the data collected from the experiment, the throughput would give the best indication here, since access is an essential part of indexing. Looking at the results from the previous chapter the best algorithm here is LH*, as it provides fast computation of document destinations. It must be noted that this is based on the setting and simplifications for this thesis. In this setting the access of documents will mostly be affected by the time it takes to compute the address. As noted earlier in the report, the affect queries have on performance is not taken into account. Had this been the case, the access speed would also be determined by how much traffic a node has, and this is again partly determined by load balance. As we have seen from previous chapters, the LH* algorithm suffers from far worse load balance that e.g. RUSH$_R$, SCADDAR and hashing/range-partitioning. The load balance also has an affect on throughput of querying, where skewed document load will lead to difference in querying speed for the nodes.

When focusing on the frequency of new documents, it is not only access speed that is important, but also what happens when scaling the cluster. If we first look back at Figure 7-21 showing the amount of documents added between scaling, we see here that RUSH$_R$ has a steady decrease in documents added. The amount of documents added determine how often the cluster needs to scale, and scaling is not a process that one would want to do to often. As the amount of added documents is very variable for two algorithms LH* and EH*, it is hard to assess the frequency of scaling.

Another element of scaling is transport volume. This is a case where the algorithms clearly differ. First scaling with hashing/range-partitioning shows terrible performance; the total volume is a multiple of the smallest transport volume for some cluster sizes.

$RUSH_R$ and SCADDAR keep an equal volume with linear increase in volume. The two methods LH* and EH* which scale by splitting, stands out with clearly the best performance with a constant number of documents moved during scaling. So looking at the scaling process as a whole, we see that while LH* and EH* can have periods where scaling happen a bit often, they compensate by doing the scaling much more efficient.

In overall performance the best method seems to be LH*. This does not mean that the other ones are not suitable. Which method which is best, will be determined by the workload of the cluster. If we look at hashing/range-partitioning as an example, we see that the large transport volume during scaling makes it a bad candidate for systems with variable data volumes. Used for search engines that rarely add or remove servers this method would on the other hand be more suitable, especially when taking into account that hashing/range-partitioning gives a near perfect load balance which affects query throughput.

What the results has shown is that, most of the methods discussed in this thesis would gain performance by scaling in batches by adding several nodes at a time. For LH* and EH*, this would mean that nodes reaching its capacity simultaneously will split at the same time. We thereby do not get very small periods between scaling. $RUSH_R$ and SCADDAR would benefit from this setup by reduction of the computation of node addresses. These two methods are also designed for this setup, where several nodes are added at the same time. During the experiment we also observed that for the methods where transport volume during redistribution varied with the cluster size, the time between scaling decreased for each round. To prevent this trend going too far, adding multiple nodes would spread the documents more, get less documents pr node and thereby get longer periods between redistribution.

9. Future work

There are a few things not included in the work with this thesis which would be relevant in a future study of data distribution. Firstly, with the implementation and testing of the methods, the scaling was not done online. That is, during scaling the indexing was paused while the documents were redistributed. Testing with online scaling would give an indication of how scaling directly effects throughput, and it would also show how the various measures presented here would affect performance when measured in isolation.

Another addition that would be of relevance is scenarios with querying included. This could show how the document load balance actually affects the query performance. Testing with queries would also show how diversity in documents types effect the performance. Among the thing one could look at here is testing with different documents sizes. If large documents get concentrated a few nodes, these will become full earlier than others and do not get fully utilized. In addition if one incorporated querying one could also see the effect of concentrating popular documents.

The RUSH family of algorithms, one of the methods evaluated in this thesis, is specifically designed for heterogeneous clusters. A further study of interest would be to test the methods on a cluster of this sort, giving some nodes more importance than others. By doing this study, the other methods would have to be adapted to handle this as well.

An interesting experiment discussed during the work with this thesis was testing of different methods of sending documents. The normal way of doing this is to send the raw data between nodes. The receiving node must then parse and index the new document, in addition to storing it. An alternate way introduced was to let the sending node retrieve its index information on the document which is transported, and send this with the raw document data. With the alternate method there is the additional disk i/o at the sending node, but less cpu-time at the receiving node. Testing this was originally incorporated into the experiment, but it was later let out because the experiment used a Lucene built index. Indices built with Lucene are composed of multiple files. To be able to test the alternate transport method, we would have to retrieve elements from multiple files. This was operations that would be very complex to incorporate into to experiment and therefore let out of the experiment. Testing the different methods of transportation would still be an interesting experiment. The best way to implement this might be to build an inverted index on its own, and thereby make it easier to modify the index elements.

10. References

ANDERSON, T. E., DAHLIN, M. D., NEEFE, J. M., PATTERSON, D. A., ROSELLI, D. S. & WANG, R. Y. (1995) Serverless network file systems. *Proceedings of the fifteenth ACM symposium on Operating systems principles.* Copper Mountain, Colorado, United States, ACM.

BADUE, C. S., BAEZA-YATES, R., RIBEIRO-NETO, B., ZIVIANI, A. & ZIVIANI, N. (2007) Analyzing imbalance among homogeneous index servers in a web search system. *Inf. Process. Manage.,* 43**,** 592-608.

BRIN, S. & PAGE, L. (1998) The anatomy of a large-scale hypertextual Web search engine. *Comput. Netw. ISDN Syst.,* 30**,** 107-117.

GHEMAWAT, S., GOBIOFF, H. & LEUNG, S.-T. (2003) The Google file system. *Proceedings of the nineteenth ACM symposium on Operating systems principles.* Bolton Landing, NY, USA, ACM.

GLENN GEORGE LANGDON, J. & RISSANEN, J. J. (1978) Method and means for arithmetic string coding IN CORPORATION, I. B. M. (Ed.).

GOEL, A., SHAHABI, C., YAO, S.-Y. D. & ZIMMERMANN, R. (2002) SCADDAR: An Efficient Randomized Technique to Reorganize Continuous Media Blocks. *Proceedings of the 18th International Conference on Data Engineering.* IEEE Computer Society.

HILFORD, V., BASTANI, F. B. & CUKIC, B. (1997) EH* - Extendible Hashing in a Distributed Environment. *Proceedings of the 21st International Computer Software and Applications Conference.* IEEE Computer Society.

HUFFMAN, D. A. (1952) A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE,* 40**,** 1098-1101.

LITWIN, W. (1980) Linear Hashing: A new tool for file and table addressing. *VLDB'80.*

LITWIN, W., NEIMAT, M.-A. & SCHNEIDER, D. A. (1996) LH* - a scalable, distributed data structure. *ACM Trans. Database Syst.,* 21**,** 480-525.

RISVIK, K. M. (2004) Scaling Internet Seach Engines: Methods and Analysis. *IDI NTNU.* NTNU.

WITTEN, I. H., MOFFAT, A. & BELL, T. C. (1999) *Managing Gigabytes: Compressing and Indexing Documents and Images*, Morgan Kaufmann Publishers.

ZIV, J. & LEMPEL, A. (1977) A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on,* 23**,** 337-343.

ZIV, J. & LEMPEL, A. (1978) Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on,* 24**,** 530-536.

ZOBEL, J. & MOFFAT, A. (2006) Inverted files for text search engines. *ACM Comput. Surv.,* 38**,** 6.

ZOBEL, J., MOFFAT, A. & RAMAMOHANARAO, K. (1998) Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.,* 23**,** 453-490.