NTNU
Det skapende universitet

Søkeapplikasjon i skyene

Stian Standahl

Master i informatikk
Innlevert: Mai 2013
Hovedveileder: Herindrasana Ramampiaro, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Norwegian University of Science and Technology

Searching In The Cloud

by Stian Standahl

Abstract

This thesis has focused on how to process and store big data in the cloud, with a special focus on challenges on creating an information retrieval system and how distributed information retrieval methods can be used in the cloud. After evaluating three cloud platforms, Windows Azure was chosen because it gave more hardware resources in the free trial than the others, and due to the fact that it had an emulator that could be used to set up the system locally before testing it on the cloud.

The search engine should also be chosen, but since Windows Azure was the preferred platform, the search engine choices was limited to those that were created in the .NET languages. I ended up with Lucene.NET because it is a powerful search tool. In addition, Lucene.NET is open source.

The evaluation was done on a distributed information retrieval system that had a server-client set up, and used partial indexes that was distributed out to the clients. The evaluation was done with a small data set to find optimization problems that has to be attended when creating a distributed system that handles large amounts of data. I carried out four evaluations on four different clients.

The results revealed optimization problems that was special for the cloud, and that has to be attended when creating a distributed system that has to process and store big data in the cloud. Also, since scaling systems in the cloud is easier, the recommendation was that scaling of the clients should be dependent on how much Azure Cache is left on the clients due to a optimization problem that has to do with the search speed of the search engine.

With some more tweaking and solving these optimization problems, the Cloud should provide an advantageous place to process and store big data.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation and General Background

When you want to create a software application, what do you need to get started? First of all there is a need for servers, and software handles the server and the application. Then, there is a need for human resources that knows how to host the servers. If the application grows or shrinks, is a requirement for scaling the resources. If it grows, more servers and human resources is required to run the application. Also, a systems developer is needed to program and design the application. On top of this, it can become quite expensive to create and host an application. This is where cloud computing makes the difference.

Cloud computing[38] is a term used for computer resources as a service. This means that in a computing cloud, the application has access to the computer resources that are virtually exposed through services and uses them as utilities. This resource abstraction allows for great scalability, either up or down, depending on the resource need. Cloud computing can do nothing about the requirement of having someone develop the application, but it can do something about needing servers or human resources to run the servers.

When creating an information retrieval application there is always a need of scalability, especially when working with big data. Big data[29] is a term used for very large amounts of data that is too big and complex to be handled by an ordinary database system. This suits cloud computing very good, since it is very scalable, and has a potential of great processing speeds. An example on big data is the data that Facebook stores away in its data warehouses. Facebook stores away 0.5 petabytes every day[20], and has had an explosive growth since they first started. In the time period from 2008 to 2012 their data warehouse has grown by 2500 times.

Since cloud computing is quite new, there is only limited of research regarding methods for information retrieval with big data. There has been done alot of research on big data that works on local servers. For example, there might be some methods that can be derived from this area. Moreover, there might also be

a problem applying these methods directly.

When working with big data there is usually multiple computers involved in processing and storing. Distributed computing[5] is a physical computer setup where two or more computers cooperate to solve one or more problems. The computers are usually physically placed in an enterprise or at a service provider, where they are managed by computer administrators. This setup allows the system developers to have a close dialog with the server administrators. The close dialog can give the developers possibility to program closer at a lower level, and lets them have better control over the computer resources. The disadvantage, with this setup, is that the scalability needs take longer to fulfill if the servers do not have enough resources. This is because the servers needs to be upgraded with new hardware. If the upgrade takes too long the application might suffer from slow performance. Moreover, the distributed systems usually are locally created on a enterprise LAN that can expose applications to the internet. Since it is on a LAN, it could be imagined that one or more applications suddenly require a much more throughput to handle the load. This scalability is something that a distributed is not able to handle well.

Cloud computing has almost the similar setup as the distributed computers. There are multiple computers that work together to solve one or more problems. But instead of having physical resources to work against, the resources are abstracted in such a way that they only can be used through service calls. This means that the applications that run on the cloud has services that are reached through services. The resource abstraction makes the application have less control of the resources. If the application is very dependent on hardware and works very closely with it, it will not be able to function well on the cloud. If an application is used for burning dvds, for example, it might not be suited for cloud computing. Nevertheless, a major advantage is that the abstraction lets the cloud computing system automatically scale the hardware resources and performance up or down, based on the needs of the application. If the application needs more processing power or memory, it will get this instantaneous and there is no need to have a server administrator physically upgrade any servers. Therefore, the application will not suffer from the distributed computing problems, for example, slow performance.

1.2 Problem Specification

A common problem when it comes to big data, is finding information contained in texts, images, videos, and other types of data. Therefore, distributed information retrieval, DIR, is commonly used for processing and storing data and since DIR works so well with big data, but has some issues with scalability. The big data processing and storing should be moved to the cloud. Therefore, an emerging problem in the information retrieval world is:

How can Big Data be processed and stored in the cloud?

Using this question as a main focus, it is possible to derive multiple research questions. For this thesis the questions are:

- How can distributed methods be used to process and store big data on the cloud?
- How well does a distributed system in the cloud perform compared to a distributed system that is not in the cloud?
- What methods are developed to meet challenges and which still remain?
- What challenges are there?
- Compared to Distributed Information Retrieval, are there any gains by using cloud technology?

1.3 Thesis Outline

The thesis is composed of chapters, starting with the introduction. The introduction will introduce challenges and problems that this thesis will try to solve. Moreover, the first chapter will give information and explain keywords that will be used throughout the report.

The second chapter will introduce alternative projects, solutions or research that are similar to this thesis. They will be explained, and key concepts that are similar to this thesis are found and introduced.

The approach chapter explains the system architecture, and the communication between the different components of the system. Furthermore, the technology used and the reason for the technology choices are explained and concluded.

In the result evaluation section, the data that is used and the evaluation setup is explained and it will lead to the results from the evaluation system. These results wll be used for discussion and conclusion in the conclusion section.

Chapter 2

State of the Art Study

2.1 Related Systems

2.1.1 SolrCloud

SolrCloud [32], SC, is a search engine that is based on Solr. The main difference is that SC can run on the cloud, and Solr can run on a distributed system, as well as a single server. The architecture in Solr and SC is built up of SolrCores which are essentially an index. It is possible to create multiple indexes by creating multiple SolrCores. In SC it is possible to have one index be made up of multiple SolrCores. Solr is built on top of the Lucene search engine, and the indexes are created by Lucene. Solr extends Lucene, and has multiple functions that makes it a good search engine. There are multiple extensions, but the function that is the most similar to what is going to be written in this thesis, is the distributed indexes function. It allows for replicating indexes over multiple SolrCores. SolrCloud is a seemingly successful system that moves distributed information retrieval to the cloud.

2.1.2 Katta

Katta[1] is a distributed data storage that is used to store any files. It can also be used to store and distribute indexes. Moreover, when files are stores on a server it is possible to have the files replicated across other servers that is connected to the system.

The way Katta distributes the files or indexes is that first the files are or index is created on a single server or on a Hadoop system (see figure 2.1). Then it is transfered to a file server, that creates index shards out of the files. An index shard is, basically, a part of a Lucene index or map reduce mapping file. The master always has an overview of the shards that are available on the file server, and will assign shards to slave nodes. The master uses Zookeeper to assign shards and control which slave nodes are online. Each slave node get the assigned shard and commences the download.

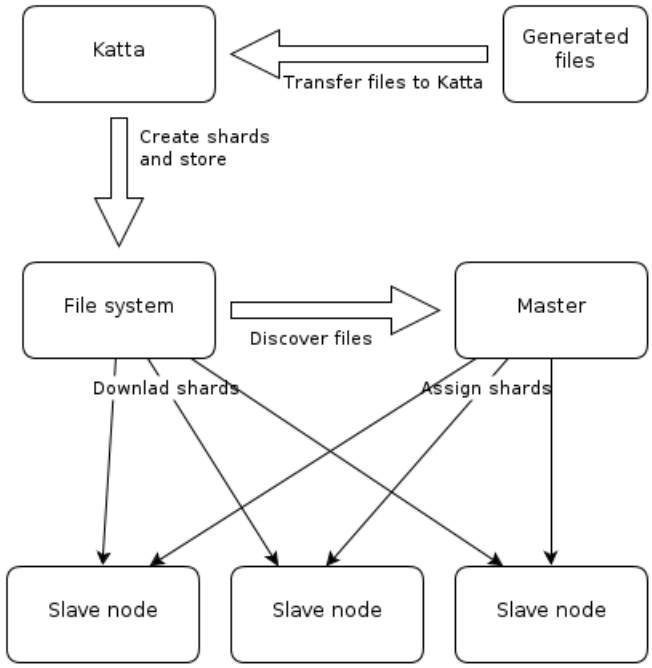

Figure 2.1: Workflow in the Katta system

The master can have a clone of itself. This ensures that if the main master fails it can automatically switch over to the clone and continue working as if nothing happened. When clients executes requests to the system, it starts a Hadoop operation, which uses multiple threads or computer nodes to solve the problem given. The similarities compared to the thesis, is the way it handles requests by using Hadoop to quickly return an answer. Furthermore, there is a similarity in how the files are stored. They are stored and distributed to slave nodes which will be something in the same direction as the thesis.

2.2 Related Research

There has been some research around the area of cloud computing, information retrieval and big data. Moreover, there has also been some research around challenges and research around cloud computing which would be interesting to discuss. This section will describe research that has been done around cloud computing on its own, and together with big data.

2.2.1 Cloud computing: state-of-the-art and research challenges

In the paper, Cloud computing: state-of-the-art and research challenges, Zhang et al. explained cloud computing in detail, and what made it different than grid computing, utility computing, virtualization and autonomic computing. They summarized that cloud computing uses virtualization technology to provide computing resources as a utility. Where there are similarities to grid and autonomic computing, but it differs in some areas.

They move on to explain how cloud computing is built up in 4 layers; hardware, infrastructure, platforms and application. The hardware manages the clouds physical resources, this can be for example a data center of servers. The infrastructure layer, also called the virtualization layer, manages the storage and computing resources. The platform layer houses operating systems that lightens the load in the infrastructure layer. It has operating systems and API that communicates with the applications. Finally, the application layers contains the applications itself.

The cloud business model is explained, showing the difference between Infrastructure-as-a-Service, Platform-as-a-Service and Software-as-a-Service moving on to the different types of clouds; public, private, hybrid and virtual private.

After explaining important products that provide and technologies that use cloud computing, research challenges are brought up. Because of its early stage, cloud computing has some challenges that should be considered. The challenges mentioned is focused on problems in the cloud computing environment itself. Some of these challenges are:

- automated service provisioning
- virtual machine migration
- server consolidation
- energy management
- traffic management and analysis
- data security
- software frameworks
- storage technologies and data management
- novel cloud architectures

Since the focus is on big data and processing some of the important challenges is the data software frameworks, storage technologies and data management, and novel cloud architectures.

The software frameworks that process and manage big data fits quite nicely in the cloud. Specially Hadoop[22], which is derived from MapReduce[19][1] created

[1] An explenation with example can be found in section 3.2.1.

by Google, is an important framework. Some challenges are optimizing the performance of Hadoop jobs and adaptive scheduling in dynamic conditions. Moreover, another challenge is to make MapReduce framework more energy-aware

The main challenge with storage technologies such as MapReduce does not use normal file systems, they implement their own. This causes an incompatibility, between legacy file systems and the applications on the cloud.

Challenges around novel cloud architectures are problems with economy, communication speed between geographical locations and the data centers, and energy usage. The challenge is how big data centers solve these problems. A suggestion was to have smaller data centers spread to different geographical locations. The smaller centers are easier to cool, and less energy goes to keeping the hardware cold. Moreover, having the data center spread out, allows for applications that must have fast response time.

The conclusion was that at this time, the current technologies for cloud computing is not matured enough to realize the full potential.

2.2.2 Cloud Computing and the DNA Data Race

Cloud computing can become a great application for process intensive applications. In the paper, Cloud Computing and the DNA Data Race by Schatz et al., its just this that is brought up and discussed. A problem is that the gap between DNA sequencing throughput and computer speed is growing. Some suggestions are made to close the gap. One suggestion is to invent algorithms that better uses a fixed computing power, but it is discarded since it is not certain that there will be any breakthroughs because of its unpredictability. Another suggestion is to make processors work more efficiently in parallel. This is where the new technology cloud computing comes into play.

As Schatz et al. says "Cloud computing is not a panacea...". Some challenges that should be considered is how to transfer big data over low bandwidth network, there are security and privacy issues, and for some problems cloud computing is inefficient.

As in Cloud computing: state-of-the-art and research challenges, they mention frameworks for processing big data is important to develop and mature to successfully develop the cloud computing. This will make the job easier for the developer depending on the framework. MapReduce is brought in to the picture as a framework that works great with most programs, but does not suit others. They move on to say that the frameworks has some challenges that must be overcome when looking at parallelism working together with the cloud. Researchers might have to develop new, more complex, algorithms to do this.

After describing the frameworks and its pros, cons and the challenges, they move on to describe other obstacles that should be solved. They start out with saying that a big hindrance is how to transfer very large datasets to the cloud itself. Since, when working with the data it must first be uploaded to the cloud, before any processing can be done. One suggestion is to physically ship hard drives to cloud vendors, when the user is not connected to a high speed connection.

Another challenge that they take up is the data security and privacy. The storage and processing security depends on local policy as well as cloud policy. However, this is not matured enough as institutions and regulators work with adapting to the new technology. The conclusion is that for now, local storage is safer than storing in the cloud.

They recommend developing methods and algorithms for parallelism in the cloud, sooner rather than later. They also have some points of recommendations that should help whether or not to use the cloud for large scale DNA sequence analysis. In short the points takes into account cost, resources, network needs, and whether there are tools that exists for working with parallel framework. In conclusion, cloud computing can be a viable option if all the recommendation points are met.

2.2.3 Big data and cloud computing: current state and future opportunities

In big data and cloud computing: current state and future opportunities, written by Agrawal et al. describes challenges by developers and DBMS designers of large internet scale applications. As the other articles they explain the different cloud paradigms, for example Infrastructure-as-a-Service, Platform-as-a-Service and Software-as-a-Service.

They move on to describe design issues when building a database management system, DBMS that deals with a single large database. A scenario is that an applications may start small, but they grow to become larger over time. Application servers can handle the scaling, but the bottleneck might be the data management infrastructure. Since open source relational database management systems have a large cost when associated with enterprise solutions, they become less attractive. The result is that key-value stores, such as Cassandra, Voldemort, etc, have become more popular because of the simplicity and low cost. They describe different systems that are developed to incorporate cloud features.

A different domain that is important for data management in the cloud, is supporting a large number of applications that have small amounts of data or also called multi tenant systems. They go through how systems work in the cloud, for example in Salesforce.com, the different applications share the same database table.

To make a good DBMS in the cloud, there are some major problems that should be looked at. The first problem that is brought up is how to provide support for ad-hoc querying on top of a key-value store or to provide consistency in systems that have different granularities. Moreover, one problem is to create more features and extend the key-value store to support more applications. However, with relational database a problem is how to make the systems utilize available resources and reducing costs. The multi tenant systems' problems focus on security, scalability, elasticity and autonomics of the resources that is in the cloud.

Chapter 3

Approach

3.1 Technology Background

When creating an application there needs to be a plan on what kind of environment the application will work in, what programs and which language is used to create the application. In section 3.1.1 a few of the most popular Platform as a Service, PaaS, providers [6] will be discussed. The conclusion will be drawn from the discussion and the application will be based on the concluded PaaS provider.

3.1.1 Choosing Platform-as-a-Service provider

When choosing PaaS provider, it is important to look at what the needs are for a certain application. For this project there will be need for many functionalities, but the most important parts are the search engine itself, and the indexing part of the search engine. The search engine needs processing power to search and index fast. Moreover, it needs a scalable storage that can handle large indexes.

When files or objects are to be used often it is normal to store them in memory. But, when creating a web application on the cloud it is not certain that the same files or objects are to be found in each call to the web service. This is because of two reasons, HTTP is a stateless protocol [21], which means that each request is treated as separate, independent transactions and each request is unrelated to other requests. Therefore, communication done using HTTP protocol is only done by request and response. Because of this statelessness, web applications in the cloud do not guarantee that the same server will process each request. This is done to have as high uptime as possible so that the web application will not become unavailable. If there is a need to hold information between each call there is usually ways of doing this, where some are session, cookies, caching, database storage, etc.

Because of the storage and processing needs, there will be a special focus on this in the discussion about the different providers, especially, the file size and the number of files that can be stored. There will also be a focus on which providers has a free subscription, and how much these subscriptions will give in computing

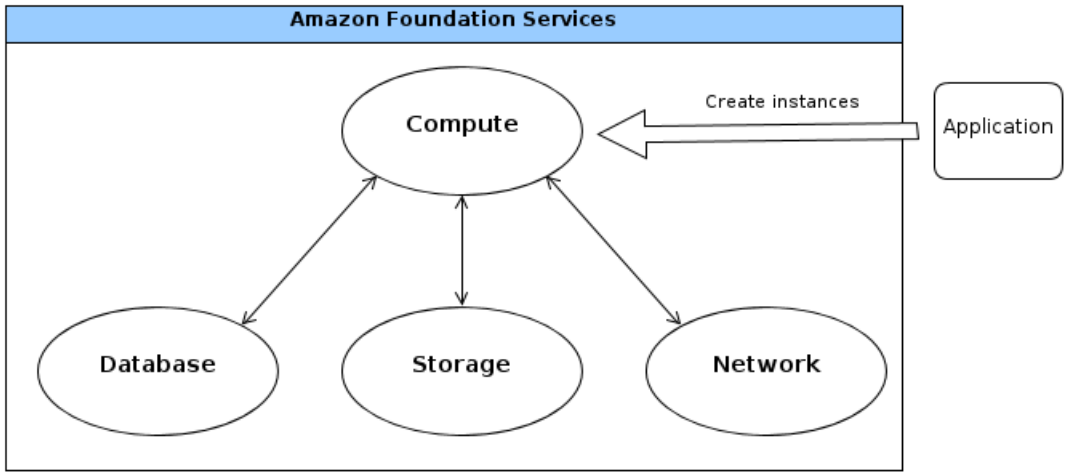

Figure 3.1: Amazon foundation services

power and storage. Most of the other services that the providers offer will be briefly mentioned.

Amazon EC2

Amazon EC2 is based on a set of multiple web services, called Amazon Web Services, AWS [37]. The services are built up of 4 foundation services, Compute, Database, Networking and Storage.

All of the services uses a REST or SOAP interface to communicate with applications, and there are many programming languages supported, where some are C#, Java and PHP.

The Compute service allows the developer to take use of Amazons computing power. It has features like auto scaling, load balancing, and Amazon CloudWatch that allows monitoring of the computing activities on the Amazon cloud through a web interface.

The Database web services is composed of 4 sub services. One can be used for a relational database, this service is called Amazon Relational Database Service (Amazon RDS). The second is called Amazon SimpleDB, which provides basic database functions, indexing and querying.

The networking service is also composed of 4 sub services, the Amazon Virtual Private cloud, Amazon Route 53 and AWS Direct Connect. The service that could be used with search engines is the Amazon Virtual Private Cloud. The Amazon virtual private cloud lets the user create a virtual network that can be closely integrated with a business network. This service allows the business network and Amazon virtual private cloud connect through VPN, and lets the business networks resources extend into the cloud. It could be very nifty to have if the search engine has a crawler that needs to go through and index a document archive at a business.

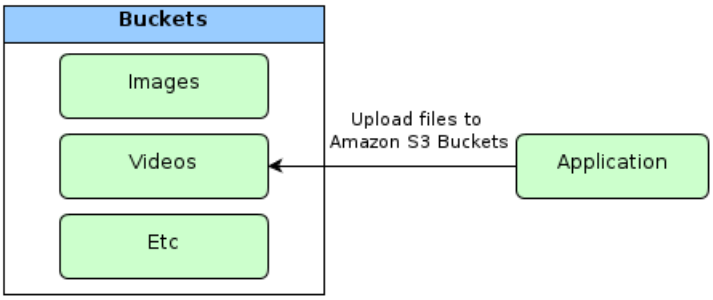

Figure 3.2: File transfer from application to Amazon S3

The most important service, for this thesis, is the Storage service called Amazon Simple Storage Service, or in short Amazon S3. This service is a scalable and inexpensive way to store big data on the cloud. Each object stored on S3 are placed into a collections called buckets. A bucket is a term used to describe a function that is similar to folders in an operating system. The bucket can have one or more data objects stored to it, and it is possible to store and retrieve the objects from it by using unique names. There is no limit on the number of objects that can be stored, and the objects can be anywhere from 1 B to 5 TB's each. The buckets have administrative possibilities from a web page that can be accessed through the Amazon web site.

The prices that this type of storage operate with are very cheap. Amazon has a free tier [36] that gives 5 GBof storage. Otherwise, if the storage exceeds this limit the fee is only 25 US cents. To develop applications for Amazon EC2, one can obtain a software development kit, SDK. The SDK is a collection of tools, documentation and examples that makes it easy to learn how to use the services.

Another important service for a search engine is the Amazon Elastic MapReduce service. This service enables the search engine to take advantage of a paralleling method that greatly enhances the processing speeds. This processing method will be explained in detail in section 3.2.1. It is a separate web service that is sorted under the Distributed Computing Services. It uses a Hadoop framework and can be combined together with EC2 and S3 services.

To speed up performance when retrieving files and objects from primary storage, Amazon has created a service called Amazon ElastiCache. This is a in memory caching service that can hold objects as if they were in memory in a normal server environment. It is used as a distributed in memory cache, where multiple applications can share the cache. The cache is scalable and has a high performance. The maximum value size is 1 MB, which is quite small if you want to store large files, values or objects.

Google App Engine

Google App Engine [25] has three ways of storing data files; Data store, blob store and Google Cloud Storage. Data store [24] is a schema less object database. It

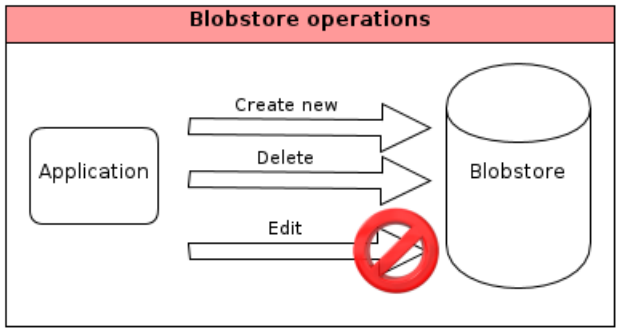

Figure 3.3: Allowed blob store operations

can store blobs together with other properties, which can be used as variables describing the blob data. One transactions can include multiple operations, and all of the transactions are atomic. This means that if one of the operations fail, it will roll back all the changes. This storage is specially designed for distributed applications, and it can handle many simultaneous transactions that process and edit same resources. The data store is also very scalable, allowing it to grow or shrink along with the activity. This storage is quite similar to a relational database, but some functions differ. The fact that its highly scalable, is one function. The problem with this storage is that the maximum file size is 1 MB and because of this, this storage is not a candidate for this thesis and will not be further investigated.

The blob store [23] is a very interesting storage alternative. It allows storage of much larger files than the files stored in the data store. The maximum size that can be read from an API call is 32 MB. The files can be uploaded or download via HTTP requests. This means that if you want to upload a file you have to create a web form, and submit the file using POST. One problem with this is that if you want to edit a file in the blob store, you have to GET the file, edit it, and then create a new blob in which you can upload to. This poses a big problem when it comes to creating indexes, and adding or removing documents from it. These operations requires multiple updates of the files, and since this is not allowed, you would have to constantly delete and create blob entries.

The third alternative to file storage in the Google App Engine is the Google Cloud Storage [26]. The Google Cloud Storage is known as a service, where the Google users can upload their files. But it also offers a RESTful API that allows developers use it as a storage for files generated by an application. When you have a RESTful API there is no need for playing around with HTTP forms to upload or download the files. To make it even better, it does not have a maximum file size. The files can be as big as you want. The storage uses buckets as object holders. Similar to Amazon, the bucket can hold one or many objects. But, everything is not as good as it looks. This storage suffers the same problem as the blob storage. The objects can not be edited. You must overwrite the object with an updated one to do an edit. But it is not as difficult as the blob store. You don't need to create a brand new bucket for each object you have to change.

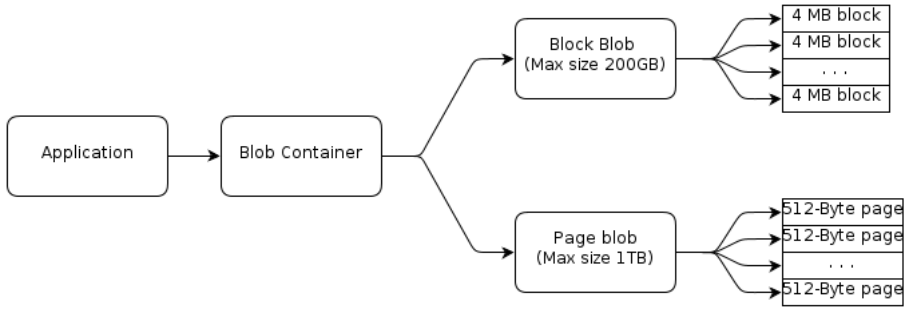

Figure 3.4: Block blob and Page blob in Windows Azure

The App Engine supports Map Reduce [27], but this is only with the Python API. So if you want to create an application with Java, you will not be able to use this feature. This is a big minus since the search indexes needs Map Reduce to process large data quantities. It is possible to use a search engine that uses python, but there are not many open source search engines that are created with Python.

As a substitute for in memory storage that is on a normal server, Google App Engine has a service called Memcache [28]. The Memcache is used to temporarily store objects so that they can be retrieved faster than from primary storage. The objects in the storage is saved as long as possible, and when it has to remove an object, due to lack of memory for example, the object that is the oldest is removed. One downfall is that the objects stored can not be larger than 1 MB.

Microsoft Windows Azure

Windows Azure [18] has two types of blob storage. One is called Block Blob, and the Page Blob [10] [13]. When you want to create a blob on this platform you create a blob container. You then have to specify which type of blob it is, block or page. With a storage account you can have unlimited amounts of blob containers, but it must be under the total size of 100TB. Each container has an unique name, and can be accessed by URL. The blobs have a RESTful API and a client library that can be used to interface with the storage.

The block blob consists of one or more blocks, up to 50 000 blocks. Each block can be maximum 4MB in size, and the maximum size of the block blob is 200 GB. Each block has an unique id, and also these can be accessed with an URL.

The page blob is not as popular as the block blob, but it also has some nice features. The random access operations have better performance. The page blob is composed of pages, where a page is a 512-byte chunk. If you want to perform a write operation on a page blob, you can either write over only one 512-byte chunk, or many chunks up to 4MB at a time. This allows for parallelization. The page blob can be maximum 1TB.

But how have Windows solved the problem of not being able to have objects in memory? They created a service called Windows Azure Caching [8]. This is a cache that is a part of the cloud application and is not shared with other applications.

There are two types of caching services; co-located and dedicated. The Co-located cache is a cache that shares the memory of an already existing cloud instance in the deployment. The dedicated cache is its own instance that can keep a cache in its own memory. The other instances may share this dedicated cache.

Conclusion

When creating a search engine the most important thing is computing speed and a large storage. The computing speed is needed to swiftly process documents for indexing, and being able to return a search result to the user as fast as possible. When it comes to search speed, a tolerable wait time for a user is 100 milliseconds [33]. This is the time it takes for a user to feel that there has been some waiting for the results to arrive. Nevertheless, Nielsen also says that if there is a loading icon present a user can wait 2 to 10 seconds, before starting to find other things to do. Therefore, the goal should be to make the search performance fast enough that the user will not feel any wait time. This means that the storage also should be fast, and should allow for large indexes. These features are very important, but there are also some features that should not be ignored. There should be a simulated environment that can be used to develop the application without having to start using the actual cloud. This will allow for testing of the application on a local computer and that could save you from accidental overuse of storage and waste of computing time.

As seen in table 3.1 all of the platforms have a maximum file size that can handle all of the storage that this project needs. Windows Azure and Google App Engine has unlimited computing hours, but Amazon has 750 hours per month available. The application would not nearly have use up all of these hours since it not will stay active all the time throughout a month. This is because it will not be a need for that much computing time. This means that computing given by all the platforms satisfy the applications expected need. The difference here is the performance. The Amazon EC2 Micro instance does not give a lot of processing speed. This instance is designed for applications with low throughput. This is quite low compared to what Azure and App Engine can offer. But the App Engine has a limitation on the amount of hours. Therefore Windows Azure is the best choice when it comes to computing power.

When it comes to storage, all of the PaaS' has sufficient storage space available. Windows Azure block storage has a maximum limitation of 100TB, but this application will not even use a percent of this space. Therefore the maximum limitation is not a problem. Moreover, the different ways of storing are also similar. All of the methods have some sort of container in which you store files. Google App Engine and Amazon S3 uses the term bucket, and Windows Azure uses the term block blob. All of these storage methods are quite similar, the buckets or block blobs are retrieved using an unique ID, and all of them have a simple way of creating and storing objects. But when it comes to how much you are allowed to store using the free subscriptions Windows Azure is a clear winner. Where the two other platforms has 5 GB of storage, Windows Azure has a possibility of storing 35 GB of data and can perform 50 million transactions out and in from the storage.

	Amazon EC2 S3	**Google App Engine Cloud Storage**	**Windows Azure Block Storage**
Storage			
Max size for one file	2 TB	200 GB	200 GB
Max total size	Unlimited	Unlimited	100 TB
Max value size in the cache	1MB	1MB	8MB
Free subscription gives			
Computing	750 hours per month, Micro instance	28 instance hours per day	2 small compute instances, 225 GB local storage each, 1 Core, Unlimited hours
Database runtime	750 hours	N/A	Unlimited
Database size	20GB	1 GB	1 GB
Database transactions	10 million I/O's	50 000/50 000 I/O's	Unlimited
Storage size	5 GB	5 GB	35 GB
Storage GET/PUT	20 000/2 000 transactions	25 000/2 500 transactions	50 million transactions
Total bandwith out	15 GB	25 GB	8 GB in, 8 GB out
Simulated Runtime	No	Java development server	Windows Azure Compute Emulator

Table 3.1: Summary of PaaS providers

5 GB is not enough for a proper benchmarking and if used it would scale enough to give a proper test result. Where as, 35 GB should be enough for testing and evaluating. Furthermore, during development there should be an simulated runtime environment where it is possible to run the application locally. This is to be able to test specific functions fast without having to deploy to the cloud. Google App Engine and Windows Azure has these test environments, where Amazon EC2 does not. The cache sizes are also important to have an greater performance is the cache size, and when the value size is larger you can temporarily store larger files, lists, objects, etc. Windows Azure has 8MB which is 8 times more than Amazon EC2 and Google App Engine with 1MB.

Since all of the platforms has quite similar functionality, there are some features that stand out which leads to a platform conclusion. This is the amount of storage possible with the free subscription, the possibility of starting a simulated runtime environment and the maximum value size in the cache. Therefore, the platform that is used in this thesis is Windows Azure.

3.1.2 Choosing a Search Engine

When it comes to open source search engines, choosing them can be a difficult task. Because, finding the right means sifting through a myriad of them. Nevertheless, after concluding with using Windows Azure as a PaaS, the choice is greatly reduced.

Since, Windows Azure runs on the .Net framework, the programming languages are limited to the available .Net languages. .Net uses Common Language Runtime [9], CLR, where the languages are translated to CLR before it is ran in the runtime environment. The CLR code is actually called managed code when it is in the runtime environment. This code is integrated to the CLR services, which are cross language exception handling, cross language integration, security and automatic memory management. The most popular .Net languages are:

- C#
- Visual Basic .NET
- C++ (Managed)

Since C++ is not preferred when creating web services, web sites, and fully taking advantage of the .NET libraries, the languages that are preferred are C# and Visual basic.

There has been done a lot of comparisons on which search engines are the best. The best, meaning, the search engine that is the fastest when it comes to indexing and searching, and how well they can find the relevant documents. Therefore, it is important to evaluate the seach engines before picking one. In the following section search engines that are found will be discussed and one will be chosen as the search engine to be used in this thesis.

Search Engine Evaluation

Evaluation of search engines is a task that takes a long time. Therefore, the search engine-choice will be based on thorough reports that have done comparison of the most popular ones. For this thesis, the most important criteria for a search engine is how easy the code is customizable, how well they retrieve relevant documents, and how fast the indexing and querying is done.

The source code for the search engine should be open. This way, it is possible to have full control over the different components that are needed. For example, if one wants to change the way the search engine does ranking. If the a search engine is not open source the querying performed will probably be done black box, which means that the query is inserted and the result is returned without knowing what is done in between. In this case the customized ranking has to be done after the results are returned, which is inefficient.

How well a search engine retrieves relevant documents are most commonly based on two evaluation measurements; precision and recall [30]. Lets say you perform a search and you retrieve a search result. The precision, P, which is the fraction of documents retrieved in which are relevant for a query, can then be calculated using this formula.

$$P = \frac{D_{rdr}}{D_{ret}}$$

Where D_{ret} is the number of retrieved documents, and (D_{rdr} is the number of relevant documents retrieved. The recall, R, is the fraction of relevant documents retrieved from the document collection.

$$R = \frac{D_{rdr}}{D_{rel}}$$

Where D_{rel} is the number of relevant documents in the total collection.

When measuring the speed or performance of the search engine, it usually is done as a black box test. The two features that is important to test is indexing and searching. The test inserts the query, and starts the timer. When the search result is returned the timer is stopped again. The search should not take longer than 100 ms [33]. The indexing is usually measured in minutes, and time it takes will depend on the number of documents and the number of words each document contains. The bigger the size of the document collection is the longer the indexing will take.

When performing an evaluation of a search engine, it is important to have a standardized document collection which is broadly used amongst other tests, to be able to compare the search engines with each other. Therefore there has been created and maintained standardized document collections that are used in the tests. There is a workshop called Text Retrieval Conference, that has a main focus of generating test collections that can be used for different information retrieval software.

As said before, search engine evaluation is a very time consuming task. Therefore, the search engine choice is based on other reports where the evaluation has been very thoroughly done. In A Comparison of Open Source Search Engines, by Middleton and Baeza-Yates, there have done an extensive evaluation of 17 different open source search engines. One important evaluation they have done is evaluating the performance of the search engines using different size of document collections, in both indexing and searching. Moreover, they look at the amount of space the indexes take after indexing is performed.

To perform the evaluation they used a document collection from TREC-4, that consisted of documents from various news paper, which in total was 2.7 GB(5572 documents). A second document collection, called WebTREC, was also used to evaluate how the search engines handled large document collections. WebTREC was 10.2 GB, and had 1 692 096 documents, divided on 5117 files.

The search engines that are not compatible with .NET and the Azure environment is filtered away, and the remaining search engines are

- Lucene
- dtSearch

Lucene

Lucene is an open source search engine, originally created in Java. After growing in popularity, there has eventually been created a parallel project that has rewritten in C#.NET, which is called Lucene .Net. This project has all of the features that the

original Lucene has. Since the project is open source, the source code is available and can be utilized and rewritten to solve specific tasks.

For index storage, Lucene uses file storage. It is possible to add and delete documents from the index without having to perform a complete re-indexing. Moreover, Lucene has searching features for example phrase, boolean, and wild card search, fuzzy searching, stemming and ranking.

When indexing the collection that had 2.7 GB worth of documents, it spent 1 hour 1 minute and 25 seconds, which was the slowest in the evaluation. But, in return it is very thorough in index compression. The interesting thing is that when Lucene was presented with the WebTREC collection that was on 10.2 GB, it spent 7 times longer than a expected linear increase in indexing time.

The 2.7 GB document collection was compressed down to 0.7 GB, which is 26% of the collection. Compared to the other search engines this was the second best compression.

Lucene did quite well when it comes to search time. It only spent 21 ms retrieving hits. It was amongst the fastest ones in the evaluation.

dtSearch

Another search engine that has port to C# is the dtSearch search engine. dtSearch is a commercial search engine where licenses can be bought to allow the use of it. There has not been any formal evaluations comparing it to other search engines, but from the web site[1] it says that it can handle over a terabyte of text in an index, where the indexing time remains under 1 second.

Since the search engine is closed, the code is not open source which means that if you want to customize the ranking, how the indexing files are stored and accessed or any other part of the system, it will complicate things and make the system a lot less effective. For example, if you want to perform your own ranking on the documents in the index, you must create the algorithm and run it after the search engine itself has retrieved the hits and ranked them.

Since dtSearch is not open source, and there are no good evaluations or comparisons with other search engines this is not the search engine of choice for this thesis.

3.2 Theory

3.2.1 Map Reduce

The Map Reduce method[19] is used to process and/or generate vast amounts of data, over many parallel computers, or computing clusters. To start out explaining the method there are two roles that needs to be explained. The master and the worker role (See figure 3.5). The master is a delegator, and distributes key/value pairs to the workers and the workers are used to process data.

[1] `http://www.dtsearch.com`

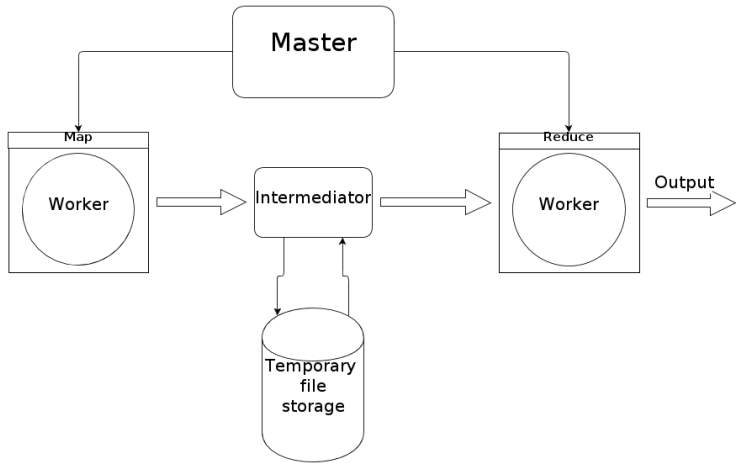

Figure 3.5: Map Reduce

The method is split into two parts the mapping part, and the reduce part. The map part takes an key/value pair and performs a process that leads to a intermediate key/value pair. The master finds all key/value pairs, and then distributes N number of pairs to N number of workers. The workers then process the values, in parallel, and returns it to the master or sends it to an intermediator.

The master or intermediator will then pass on the the intermediate pairs, and group all the values belonging to one key. Then the master will distribute the keys together with the grouped values to reduce functions. The reduce functions will merge together the values so that the output from the reduce function is either null or one value. The intermediator is a function that is used to alleviate the dependency on storing huge amounts of data in memory. To better explain this method the following examples are created.

To give a real world example, Map Reduce can be something like an organization. Where the master is the leader that gives orders to a worker. The order could be something like: given this word and these documents, count the number of occurrences of the word in the documents. The worker does this and gives the result back to the leader. The leader could give the same order to multiple workers with the same word and different documents. These workers would also return the results to the leader. The leader would then group all of the results and give the order to a worker to sum up all the results for a word that is given together with the result list. The results from the summation will also be returned to the leader, and the leader will combine all of the results into one list. This list is the finished list and now the leader has the occurrences of all words he/she asked about.

3.2.2 Information Retrieval

Information retrieval, IR, is the study of finding relevant information in unstructured data [30]. The unstructured data can be pictures, video, audio, or text,

but since this thesis only uses textual IR, this will be the focus. Generally, there are multiple steps that lets a search engine perform IR. There are two steps that must be in place before a search engine can perform IR. One step is to manipulate the unstructured data in such a way that it allows for the most efficient IR. This is called indexing. The second step is actually performing the IR, this is called searching or querying. There is also a third step that should be included, but this step will not be explained in detail. This step is called crawling, which will be shortly explained at the end of this section.

Search Engine Architecture

As explained in the introduction to this chapter, the search engines are composed of multiple parts. To explain the architecture of the search engine, it is nice to have an example that explains the general parts of the system. Lets say you have a user, that has a need for information (see figure 3.6).

To let the user find anything at all the search engine must have an index. The index is created by either adding documents manually, or there can be a crawler that automatically crawls the internet and adds each unique web page as it finds them to the index. In this case the user gets to upload the documents. When the user uploads a document, the document will go through a number of processes before it gets searchable.

First of all the document must be sent through a document analyzer. The analyzer will retrieve the text in its plain form with no styling or formatting. Then the text will undergo something called preprocessing, which involves removing stop words and symbols, converting all the characters to lower case characters, and

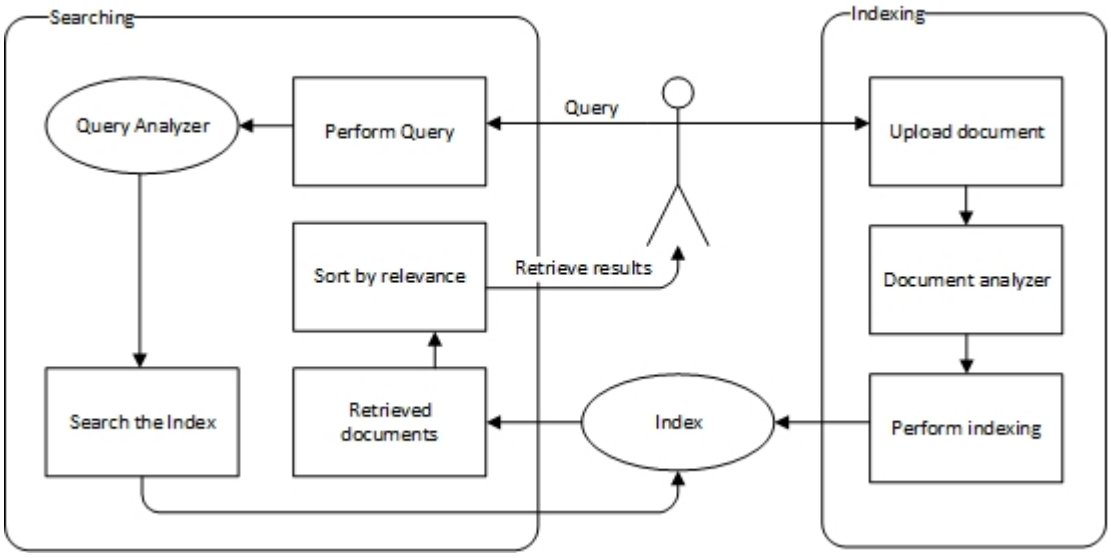

Figure 3.6: A general search engine architecture

stemming. Stop words are words that are repeated so much that it does not give any meaning searching for it. Example of words are: and, are, is, he, she, etc.

As said before, the words must also undergo stemming. This is a process where the word gets suffixes removed, and the words are returned to its base form. Cows are turned into cow, running is turned into run and broken is turned into broke, just to mention a few examples.

All of this preprocessing is done to make the rest of the processing easier, faster and to save storage space. But the preprocessing can create some problems for later searches. Lets say someone wants to search for the phrase "He or she likes cake". Because of the preprocessing the words he, or and she are removed and will not give any hits, and will be removed automatically from the query, giving the phrase "like cake". A lot of the semantic meaning with the sentence is now removed and the results might not be as good as the user hopes. Therefore, common phrases should be carefully treated.

After the preprocessing of the texts, the index is left with a lot of processed words called terms. To be able to perform fast and efficient searches these terms can be stored in multiple ways; either in an inverted index, bitmaps, or they can be used to create a probabilistic model. The most commonly used storing method is the inverted index, which is a list of words where each word has a list of which document it occurs in. Since the thesis does not involve details around how the indexes are stored, there will not be much explanation done around it.

Back to the user that has an information requirement. What is the next step when the requirement is converted to action? The user performs a search with words relevant to the information topic. The query undergoes the same preprocessing done when indexing, and then it is sent to the index for document retrieval. All of the documents that contain the query terms will be retrieved. If the search is a boolean search, the documents are returned with no relevancy ranking. But usually, before the documents are returned to the user, they will be ranked by relevancy where the most relevant document is at the top and the least relevant document is at the bottom. Some ranking functions are cosine similarity and Okapi BM25[2].

3.2.3 Distributed Information Retrieval

A distributed information retrieval system [3], DIRS, is an IR System that consists of multiple nodes that work together to solve information retrieval problems. A distributed system is very good at storing and process big data, but this assumes that the computers in the system is physically placed in the same room or building, and that the system is able to handle the amount of processing power and storage space needed to handle the big data. A problem with a distributed system is that the communication between the computers is slow compared to one single computer. Therefore, communication between nodes must be as low as possible to fully utilize the potential.

[2]More information about these ranking methods can be found in Introduction to Information Retrieval by Manning et al.

If the amount of data is not as big and complex, there is not a need for a distributed system. A distributed system should be slower to process a small data set than one computer, because of latency on the network is slower than a computer bus. When it comes to scaling a single computer can not scale as well as a distributed system. Moreover, a distributed system can handle a much larger data set than one single computer.

There are two general ways how the distributed system can be designed. One way is to have a centralized system, where the computers are set up in a client-server relationship. The other way is to have a decentralized managed system, also called a peer-to-peer system.

Centralized System

The centralized managed system is composed of one master server or master node that is in charge of partitioning the document collection to one or more slave servers or slave nodes. The master node will at all times keep track of which slave nodes are online at all times, and will partition the index in specific ways that suits the information retrieval needs.

One way to partition and distribute the index is to create one index that covers the whole document collection, and creates duplicates that are distributed to all the slave nodes. This design allows for automatic rerouting of the requests if one slave node goes off-line. Moreover, it also allows distribution of the request to all the slave nodes so that they get the same amount of traffic, also called load balancing. Viewed from the users point of view, the system will have a stable search time and performance. Viewed from the systems point of view, there will be less processing peaks and one slave node will not be overwhelmed with requests.

A problem with this design is that it does not allow the system to utilize the power of parallelization. Lets say the system uses the index duplication and wants to perform a query request to all of the slave nodes, in parallel. All of the slave nodes will receive the requests and each slave node will return the same result. This means that if each result set is put together, it will still be the same as the result from one slave node. Therefore, performing queries in parallel is rendered mute, and one request to one slave node should be sufficient to get the correct result.

When it comes to indexing it is essential that all the indexes are the same. Therefore, the indexing must be a broadcast to all of the slave nodes, which means that parallel indexing is essential.

The time it takes to create an index, will be the total time it takes to perform an indexing on a single computer, and the time it takes to transfer the index to the different nodes. The time it takes to transfer the index, is dependent on how this is performed. One way is to transfer to all of the nodes at the same time, this will strain the bandwidth and will slow down the transfer time from the master node to the slave nodes. In this case the total time, T, will be the time it takes to create the index, $T_{indexing}$, and the time for the last node to complete the transfer, T_{last}.

$$T = T_{indexing} + T_{last}$$

The search time will be the time it takes to perform a search with a normal

search engine. In addition, the time it takes to transfer the query from the master node to the slave node and the time it takes to transfer the search result from the slave node to the master node.

The time it takes to perform partial indexing will be the time it takes to transfer the documents to the slave nodes and the time it takes to for the slowest slave node to finish indexing. Since, the slave nodes will have less documents than the total document collection, the indexing time for each slave node must be lower than indexing the full collection.

There is also a different way to partition and distribute the index. The index can be split into partial indexes that is distributed to the slave nodes. There are two ways a partial indexing can be achieved; either by local indexing or global indexing that is compared in an article [34] written by Ribeiro-Neto and Barbosa.

Local indexing is a partitioning method, each machine is its own search engine. Meaning, each machine queries and indexes without the knowledge of the other machines. Because of this separation this method is great at load balancing, since the queries can be distributed to all the machines in parallel. But, the document frequencies and other global information is not easily obtained without having to perform a post indexing task, where the global information is retrieved.

Global indexing is when a large index is split up by term and distributed to different machines. In this case, the machines is not its own separate search engine, and the information distributed is global information.

The comparison done by Ribeiro-Neto and Barbosa found that global indexing was favored over the local indexing when it comes to query processing speeds in a tightly coupled system. The network system was a fast switching network where there were fast message exchange among the computers, which will reduce the networking time and in turn reduce the query processing time. They also mentioned that the average precision, in a local indexing system, drops as the number of machines increases because of the lack of global information. To combat this the recommendation was to gather the global information and use this in the results merging process.

When using a distributed information retrieval system with a higher networking time factor, Cambazoglu and Aykanat found in [4] that a local indexing method was preferred when there were infrequent queries to the system. But, when there were frequent queries the global indexing method was the preferred method. Furthermore, they found that the global indexing method was slower when it came to infrequent queries, because of disk access and network volume imbalance between the machines.

When the master node performs indexing of the document collection, it must at all times keep track of which slave node has what partition of documents. Moreover, the indexing must be done in such a way that if a slave node goes off-line, it does not affect the query result. It must always return the correct result. One solution is to let two or more slave nodes get the same partition of the index and when one slave node goes down, the other takes over. The other way is to have the documents partitioned in such a way that the indexes have overlapping documents. This technique is very similar to the RAID 5 technique, where three or more hard disks

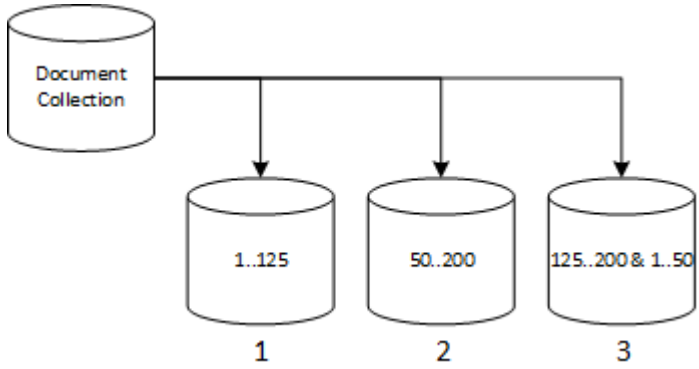

Figure 3.7: Document partitioned with overlapping documents

work together as one logical unit. If one hard disk fails, it can be replaced with another without the loss of data.

An example of partial partitioned indexes can be seen in figure 3.7. In this case the documents are distributed amongst three slave nodes. The first slave node will get the documents with numbers from 1 to 125. The second slave node will have the numbers from 50 to 175, this means that the documents from 50 to 125 will be duplicated on the first and second server. The third slave node will have the numbers from 125 to 200 and 1 to 50. The third slave node will duplicate the documents from 125 to 200 and 1 to 50.

All the documents will now have one duplication on another slave node, and if one slave node goes down the query will still be able to search through the whole document collection, and return a correct result.

In both the partial index and the clone technique there is a need to keep all the slave nodes updated with which documents that actually exists in the document collection. It is possible to do a full indexing for each update, but that is a big waste of computing resources. In the case of cloned indexes it is possible to do a broadcast to all the slave nodes to get them to update themselves. To update the partial indexes it is also possible to do a broadcast, but here only the slave node that has the relevant partial index will update itself, and the rest of the nodes will ignore the update.

The documents can be distributed amongst the partial indexes in a few ways, where the most important ones are the sequential, randomized or semantically grouped method.

The sequential method will just go through the list of documents one by one, either by alphabet or by id, and pass the same number of documents to each slave node. This does not take into account the file size or text size of the documents. In worst case, one slave node can be stuck with a all of the largest documents in the collection, and the rest gets only small documents. Therefore, this method can be very uneven in the distribution. Nevertheless, this method is easier to implement than the other.

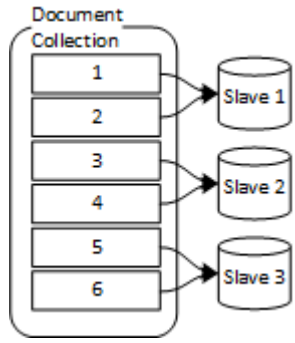

Figure 3.8: Partial index with sequential partitioning.

The randomized method will distribute the documents to the partial indexes in a random way. The method can do it totally random, but this will not be any better than the sequential method. The documents will then be distributed based on their size. When a query is performed each slave node will need to process a similar amount and they will use a similar amount of time.

The next method takes it one step further and groups the documents by semantics. The documents can already be categorized so that each slave node holds one category of documents, or the documents can be automatically categorized or clustered.

Decentralized System

The decentralized system is where all the search nodes work independently, and the document collection is uncontrollably partitioned amongst the search servers. Lets say you have a initial node that receives a query. This node will then query all the other peers on the network and retrieve a result. The initial node will then merge the result and return it to the user. In this case the it is difficult to perform a query to all of these servers without having a central managing component. The nodes must at all time have a way to reach all the other nodes, and must also know which nodes are online. Moreover, if one node goes off-line, the other nodes must have enough information to still compose a full index of the document collection. One of the key components in this type of system is the network connecting the nodes to one another. The network must be fast and reliable to have a successful system.

The other way is to create partial indexes and distribute those to the slave nodes. This enables the user to perform a parallel query to all the slave nodes. The query can be faster because of smaller partial indexes and the parallelization. The master node must keep track of what files are in what index, and if all the partial indexes together represents the document collection. One problem in this design, is the fact that when there has been a query to the slave nodes, all the slave nodes will not know of the queries to the other slave nodes. When the results from the slave nodes are are gathered, the master node must merge the result and then

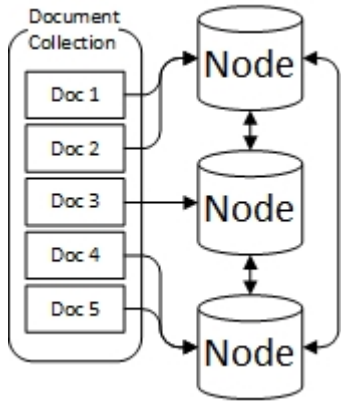

Figure 3.9: Peer-to-peer distributed information retrieval

return it to the system performing the query.

3.3 System Architecture

3.3.1 Overview

The system is built up using a centralized distributed system, which is composed of two main components; the master node, and the slave nodes. The master nodes is responsible of knowing which slave node is active at all times. Moreover, it is important to have node responsible of knowing which files belong to whom. Therefore, the master nodes is in charge of distributing the files to the different slave nodes. As mentioned in 3.2.3 in the section about centralized systems, there are multiple ways of distributing indexes amongst the slave nodes. One way was to create partial indexes on the slave nodes. In the system the master node does not do any indexing itself but it only distribute the files to the slave nodes so that they can index the files themselves.

The slave node is responsible for creating, maintaining and querying the index. This setup not only supports one slave nodes, but it handles multiple slave nodes, but each slave node itself is its own search engine and they work independently from all other nodes. This means that each slave node can theoretically be physically placed on separate servers, but this would not be very efficient because of the network delay between the nodes.

Master and Slave Communication

The communication between the master and the slave is performed by utilizing web services. The chosen web service is a simplified Web Application Programming Interface, or usually called Web API [16]. It is also possible to create a web service using Windows Communication Foundation[17], WCF, but because of time

constraints and a higher difficulty to set up a project that uses this technology, this was the technology of choice.

The master and the slave nodes is built using the Model-View-Controller, MVC, framework [12], where the master node uses views to render HTML pages for the user, and the slave nodes only has controllers that the master node can use for communication. This means that when the master node wants to delete a document from the index, for example. It will send the the document id to the respective slave node using a URL to the slave node. The URL could look something like this `http://slavenodedomain/Service/Delete/1`, where the controller name is in this case Service, the action is Delete and the id of the document is 1.

A problem arises when this communication is done in full duplex[3]. Since the communication is over web API it is usually done through port 80. When one slave node tries to connect and for example download files, the same slave node can not perform a simultaneous call to the mater node. Therefore the communication is done in half duplex[4], where it will in most cases be a one way communication; from the master to the slave nodes. The one exception to this is when the slave node starts up and notifies the slave node that it is online.

Dataflow

In the overview it was mentioned that the master node had a web site that a user can utilize to perform searches and indexing. The flow from the web site and to the search engine in the slave node contains multiple steps that needs to be traversed before the user can obtain a search result or index a file. Generally the data flow is done like this; the user performs an action on the web site that must utilize the search engine. The request will go from the master node and either to one specific slave node or all of the slave nodes depending on the action and the amount of slave nodes. For example, if the user wants to index one file and there is not enough slave nodes to have overlaps of this file. The file will only be sent to one slave node for indexing. The master node will perform an URL call with the needed parameters to one action on the slave node. The slave node will use the search engine, and in turn the search engine will utilize the Azure blob to obtain the index and perform the action. If there will be a result from the index it will be returned to the master node, which will in turn give the result to the user.

File distribution

To be able to search in the index, the slave nodes must have indexed files, and these files have been distributed by the master node. When the files are distributed there were some important criteria that had to be fulfilled. The files had to be evenly distributed amongst the slave nodes, which would ensure an even load. Some method that could do this, which can be read about in 3.2.3, but the method chosen was the most simple algorithm, the randomized method 3.1. This method

[3] Full duplex is when there is simultaneous communication both ways between two nodes.

[4] Half duplex is when there is communication both ways between two nodes, but it is not allowed to have communication simultaneous in both directions.

```
A = whole document collection
N = total number of nodes that will be distributed to
n = 0

for each file in A
  if node number n does not have the file
    assign the file to this node
  else
    select the same file for the next iteration

  n = n + 1
  if n is the same as N
    n = 0
```

Listing 3.1: Algorithm for randomized file distribution.

takes all the files in any order and distribute them randomly to the slave nodes, not taking into account semantic meaning of the text or file size. This method was picked because of the quick and easy implementation. It is a good method if the text in the files are similar in size and word count. But, if one slave node randomly gets all of the largest file in the document collection, it would have more load than the others. Since, in a distributed system, the search and indexing is only as fast as the slowest slave node, this would have a negative effect on the performance.

Another important criteria that have to be met is that one or more slave nodes should be able to go off-line, rendering them unavailable to the master node, without affecting the total document collection that is distributed to the slave nodes. This means that the file distribution should implement a method for overlapping the files. Based on the number of overlaps needed, the same file should be distributed n number of times to n slave nodes. The percentage for uptime is guaranteed to be 99.9% for most of the services in Windows Azure [15], and if the system was based on these numbers there would probably not be any reason for having overlapping files. But, there could be some errors or one slave node could be taken off-line to perform an update on the code that negatively affects the uptime. But, if all of the files is overlapped as many times as there is slave nodes, there would not be partial indexes and there would not be any performance enhancement gained from parallelization. Therefore, the number of overlaps can be calculated from 3.1.

$$O = \lfloor \sqrt{N} + 0.5 \rfloor \tag{3.1}$$

where N is the number of slave nodes. Using this formula to find the number of overlaps for 2 slave nodes, there would be no overlapping, but if there where 3 slave nodes there would be one overlap of each file which would allow one slave node to go off-line without affecting the total document collection.

To integrate overlapping into the file distribution the randomized algorithm 3.1 would have to be run O number of times. Which would overlap each file O times.

```
1  F = number of files to be taken from the existing slave nodes and
       added to the new one
2  N = total number of nodes excluding the new one
3  n = the new node
4  Repeat F times where the counter is i
5    index = i \% N
6    take the last file from node i and add it to n
7
8  Add n to N
9
10 OB = Calculate number of overlaps based on number of nodes excluding
       the new one
11 OA = Calculate the number of overlaps based on number of nodes
       including the new one
12
13 Repeat OA - OB times
14   Run the algorithm for Randomized file distribution~\ref{
         RandomizedFileDistribution} with N
```

Listing 3.2: Algorithm for evening out the index after adding a slave node.

Adding a Slave Node

To make the system as scalable as possible, the system must automatically adjust if a slave node is added or removed.

When a slave node is added, the master node must then register node and proceed to evening out the index so that the new slave node will have files (see listing 3.2). Evening out the index will give an increase in performance, since the indexes will average out to be smaller after the slave node is added. Depending on the number of nodes, the files will also be overlapped more, making the system more resilient against a slave node going off line.

In the algorithm, F, The number of files to be taken from the existing slave node (see equation 3.2), can be calculated from taking the total sum of the files, f, in the list of slave nodes excluding the new node and dividing by the number of slave nodes including the new one.

$$F = \frac{\sum f \in N - n}{\sum N} \tag{3.2}$$

Removing a Slave Node

As mentioned Windows Azure has a very good up time for instances running on the cloud. But there might be some other reasons than a failing hardware making a slave node go off line. Just as adding a new slave node there is a need for evening out the index after removing a slave node.

Eventhough a slave nodes goes off line, the master node keeps lists of all files the node has indexed. This allows redistributing the files that belong to the slave node that has gone off line (see listing 3.3). Before the redistribution is performed, the number of overlaps is calculated based on all the slave nodes excluding the off line

```
1  F = Total file set
2  N = Total number of nodes
3  n = node to remove
4  OB = Number of overlaps based on N
5  OA = number of overlaps based on N - 1
6
7  hash = hash table
8  for each file id, f in n with counter i
9    if OB > OA then
10     add f to hash
11   else
12     add f to N that has index i \% (N-1)
13
14 if OB > OA then
15   for each file in (F - hash) with counter j
16     remove file from N that has index j \% (N-1)
```

Listing 3.3: Algorithm for evening out the index after removing a slave node.

```
1  nodes <- Get slave nodes
2  files <- Get files
3  threads <- Initialize list of threads
4  Run randomized file distribution mapping files to nodes
5
6  for each n in nodes
7    thread <- Add and start new thread that sends files to node n for
         indexing.
```

Listing 3.4: Algorithm for indexing files in parallel.

slave. If the number of overlaps is less than the number of slave nodes including the off line node, there is a reduction of overlaps.

Indexing

Before the files are indexed, the algorithm used for distributing the files(See listing 3.1), to the slave nodes, is executed. Then, each slave node gets a separate thread that sends the files to the slave nodes for indexing through the web services.

Each file that is sent to the slave node is sent as a POST that includes the text and the id of the document. When it is done in batches, a list of objects containing the text and the id is created and then serialized[5] into a JSON text[6]. When the JSON text is received by the slave node, it deserializes[7] the text into the list of

[5]Serialization is the process of taking one object or a group of objects and flatten it into a sequence of bits so that it is easier to send over a network (`http://www.parashift.com/c++-faq-lite/serialize-overview.html`)

[6]JSON (Javascript Object Notation) - "JSON is a lightweight data-interchange format. (...) JSON is a text format that is completely language independent (...)" - ref: `http://www.json.org/`

[7]Deserialization is the opposite of serialization. It is to take the sequence of bits and reconstructing the object.

objects and runs the indexing. The id and the text is kept in the index for retrieval later.

Result merging

The total file collection is distributed to the slave nodes using a document-wise partitioning (see section 3.2.3) . When a query is done to the system, the master node must query each slave node for the relevant documents. Each slave node will, in parallel, perform a query to the index and return the documents found to the master node. When the master node has all of the results it must perform a ranking.

Each search result that is retrieved contains posting lists for each query term and a term frequency for each term. The master node gets the posting lists for each term and creates a new one, where all duplicate document ids are removed, leaving an unique posting list. All of this information is used to perform a cosine similarity ranking between the ranking and each search result.

3.3.2 Caching

Caching

Each slave node have a cooperative cache (see section 3.1.1). As figure 3.10 explains, the cache is used together with the Blob storage, to speed up the index file retrieval. The index files are stored permanently stored to the blob storage, and when the index files are either stored or updated in the blob storage the cache is updated at the same time. The same with deleting, if it is deleted from the blob it will be deleted from the cache. If the file is to be retrieved and it does not exist in the cache, it will look in the blob and retrieve it from there. The cache will be update at retrieval.

There are a few problems that occurs when it comes to using the Azure cache. The objects has a max object size of 8 MB [11]. This means that if there are no special treatment of the files stored in the cache and the index files grows larger than 8 MB, they must be retrieved from the blob. Another problem is that the total size of the cache is limited to the percentage of memory that is designated to the slave node. For this thesis, the slave nodes runs on a small instance[Corp.], and therefore the cache can be maximum 1.75 GB, which would use 100% of the instance memory. But using 100% of the memory for cache is not recommended, since this would cause problems for normal procedures in the program. Therefore, the slave nodes have been set to utilize 50% of the memory for cache.

The maximum object size can be solved by dividing the file into partial files that are smaller than 8 MB, and then use a hash where the key is the filename and the value is a list of the partial file names. This hash object can also be stored since the cache can hold any object that can be serialized [14], for example CLR objects, byte arrays, XML files, etc. byte arrays This enables retrieving the object and merge it to the full file. The down side by doing this is that it might not be faster than the blob storage, since processing time for splitting and merging

the files must be taken into account. As with the blob storage there is no extra processing needed to store and retrieve the files.

The maximum cache size can not be altered except for increasing the size of the instance. However, the biggest instance has a memory of 14 GB. What if the index grows larger than this? The solution here is to add another instance, which can be done through the Azure API or through the Azure management web site, and then even the index out to the new instance.

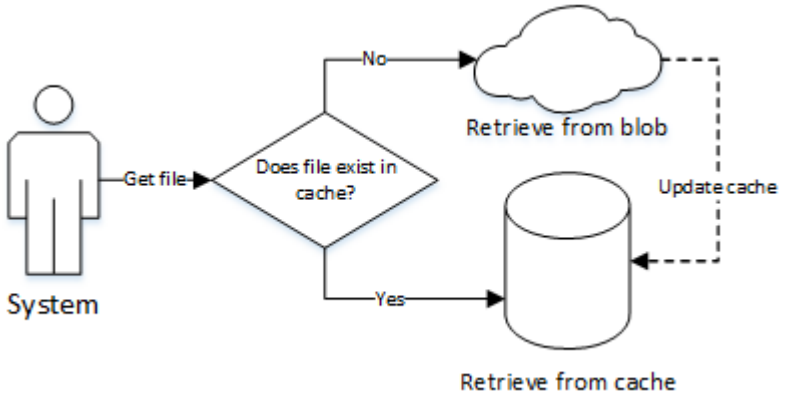

Figure 3.10: Getting a file from storage in a slave node

Therefore, to handle big data the slave node instance should scale up as the cache size is being used up. Lets say that a small instance slave node with a memory of 1.75GB, has cache size of 50%. Then the index files should not grow larger than 0.875GB. If it grows larger, each query would have to retrieve index files from the blob and from the cache. It would not be penalizes as much as when retrieving everything from the blob, but there will be an increase in query time.

Chapter 4

Evaluation

4.1 Data Set

When evaluating an information retrieval it is important to have a good data set to test the functionality up against. The data set should be large and have decent sized texts so that the system can be put under stress. But, unfortunately only parts of the dataset is used for this evaluation, because of time constraints.

This experiment will use the Wikipedia English article dataset. It is a XML that contains all the articles written in the English version of Wikipedia, and presented in plain text without any other media (f.ex. pictures). The XML contains alot of information that is needed run the Wikipedia web pages. In code listing 4.1, the data structure is written out in a stripped XML structure that contains no value in the XML elements. Moreover, there have been some elements and attributes left out to make the structure as clear as possible. The elements that will be used is the page element with the child elements; page id, text, time stamp, title, user name and comment.

The whole article collection is contained in one XML file that has a file size of 41.98 GB, and if the whole data set would be used for this evaluation the information given in the evaluation of Lucene (See 3.1.2), it seems that this file size should be more than sufficient to test the search engine. Approximately 100-200MB will be used to find bottlenecks and possibilities for optimization.

4.2 Evaluation Setup

There will be done two evaluations of the system, one with scaling and one without. The purpose of performing the evaluation without scaling is to get a control result in which the scaling result can be compared against. Moreover, the bottlenecks that is found during the system development will be specifically tested to find optimizations that can be done to the system.

[0] `http://en.wikipedia.org/wiki/Wikipedia:Database_download`

```
<mediawiki>
  <siteinfo>
  ...
  </siteinfo>

  <page>
    <title>[Title]</title>
    <id>[Article id]</id>
    <timestamp>[Time and date of last edit]</timestamp>
    <contributor>
      <username>[Username]</username>
      <id>[User id]</id>
    </contributor>
    <comment>[Comment from contributor]</comment>
    <text>[article text]</text>
  </page>
</mediawiki>
```

Listing 4.1: XML structure of the file that contains the English pages from Wikipedia

The two evaluations will be done in a stepwise manner, where the files is indexed with a fixed number of bytes at a time. Each iteration the total indexing time, search time and index size will be noted. Doing this gives a good rate of result collection and will give an overview of how these factors will change. It is also important to see how the indexing- and search time responds to increasing the number of nodes, and to find a specific trigger where this increase should be done.

The evaluations will be done on an emulator, except for the bottleneck tests. Therefore, the networking delay is a factor that can be neglected. This is because the networking delay is the time it takes to do a loop back[1]. After running a ping to the loop back interface the network delay was found to be less than 1 ms, therefore this time factor can be neglected.

Each evaluation will have the same instance configuration. Both master and slave nodes will run on a small instance[2], where the cache size is set to be 1% of the total RAM, which is 17.5 MB.

4.3 Evaluation without scaling

Each evaluation will be divided into the number of slave nodes that are connected to the master node. This section will show results from indexing and searching with 1 to 4 nodes. There will not be any scaling involved here, which means that the evaluation will have the same amount of slave nodes at the beginning as at the end. The evaluation with 1 slave node will include some extra evaluations, that reveals some important optimizations that can be done to the system.

[1] Loop back is when the network signal goes through the network card, without going past it.
[2] Small instance has 1.75GB of RAM

The results that are found in this section is a summary of the total result set that can be found in A. The focus have been to draw out specific points of interest.

4.3.1 1 Slave node

Indexing

The results in 4.1 shows the results from indexing the first 3 groups and the last three groups, using 1 slave node. The index size grows in iterations of approximately 7.33 MB, which means for each iteration, i, the index files that will be loaded from the blob to memory each time will be $7.33xiMB$. As shown, this results in an increase in indexing time.

The files are transfered at approximately 2 MB at a time, and what is somewhat interesting is that the index actually is larger than the files that are transferred. This is probably because the text is stored, uncompressed, in the index.

Group	Time (min:sec)	Size (MB)
0	01:25	3.40
1	00:59	13.61
2	00:55	20.48
...		
12	01:13	85.74
13	01:28	92.30
14	01:25	98.73

Figure 4.1: Results from running, both, master and slave node on a single computer without the use of scaling in groups of 2MB.

When running a different test and doubling the group size, the results 4.2 show that the indexing time decreases as the group size increases. By looking at the size and the time compared to those in 4.1, it is clear that the MB/s is greater. In fact, the approximate indexing throughput[3] for the indexing using 4MB is on average 0.233 MB/s, whilst the group of 2MB had 0.111 MB/s.

Searching

The data displayed in 4.3 is the results from searching after each file group is indexed. The table contains reference to which file group the results came from, the search time, and the number of hits that came from the search.

From the results it is interesting to see how the caching helps on the searching performance. For each file group there are 5 searches done. The first search in each group is significantly slower than the others. Whilst the other searches are almost instantaneous.

[3]The MB/s it takes for the files to be transferred from the master to the slave, and then stored in the index.

Group	Time (mm:ss)	Size(MB)
0	00:52	6.76
1	00:47	26.56
2	01:08	40.01
...		
12	01:31	168.72
13	01:32	182.06
14	01:28	195.03

Figure 4.2: Results from indexing with 1 slave node without scaling in groups of 4MB

Looking at the results from the four last rows in each group, it seems that the number of search hits does not affect the time it takes to perform the search. But, looking at the first row it seems that the index size have a great affect. As the size grows, it takes longer to load the index files from the blob into the cache, and therefore makes the first search much slower.

Group	Hits	Time (ms)
0	4	06.661
0	4	00.256
0	4	00.232
0	4	00.720
0	4	00.143
1	14	06.244
1	14	00.672
1	14	00.486
1	14	00.526
1	14	00.680
2	20	07.547
2	20	00.651
2	20	00.533
2	20	00.702
2	20	00.514
11	20	17.041
11	20	00.351
11	20	00.334
11	20	00.325
11	20	00.406
12	20	18.854
12	20	00.323
12	20	00.442
12	20	00.217
12	20	00.263

Figure 4.3: Results from searching with 1 slave node.

4.3.2 4 slave nodes

Indexing

The results found in 4.4, show that the result is quite similar to indexing with 1 slave node. The indexing time grows together with the index size. But, as expected, indexing with 4 slave nodes made the index approximately twice as large as the index with 1 slave node. Because of the overlapping files.

Group	Time (mm:ss)	Size(MB)
0	01:58	6.88
1	01:35	27.60
2	01:34	41.54
10	02:14	146.82
11	02:08	160.41
12	02:54	174.05

Figure 4.4: Indexing results, using 4 slave nodes.

Searching

As the results from searching with 1 slave node, the results from searching with 4 slave nodes, 4.5, show that the initial search is quite slower than the preceding searches. Also, as the index size grows, the initial search time grows.

4.4 Evaluation with scaling

4.5 Bottlenecks

4.5.1 Cache vs Blob Storage results

To test the reading and writing operations done to the Azure Blob and the Azure Cache, a program was created and uploaded to the cloud. This program was running on a small instance, with 1.6 GHz CPU and 1.75GB RAM. After running tests with increasing increments of 10MB, starting on 10MB, the cache seemed to have a smaller increase in reading and writing time than the blob 4.6. For example,when using a file that was 80MB, the cache used 67.4% less time when writing and 50.56% less time when reading.

From the graphs drawn from the table 4.6 and 4.7, one can see that Azure Blob has a much higher growth rate than the caches time. If, only, Azure Blob is used to read and write files the query time will be greatly affected, since the index files must first be retrieved before the actual query is done.

But, as the size of the files grow one can see that Azure Cache might also take to long to retrieve the files. At 80MB it takes 1.6 seconds to retrieve the file and then it must process the file to retrieve the relevant data for querying. This means

Group	Hits	Time (ms)
0	4	06.906
0	4	00.265
0	4	00.174
0	4	00.180
0	4	00.151
1	14	07.049
1	14	00.481
1	14	00.560
1	14	00.799
1	14	00.420
2	20	09.032
2	20	00.761
2	20	00.826
2	20	00.547
2	20	00.625
11	20	23.047
11	20	00.832
11	20	01.187
11	20	00.788
11	20	00.902
12	20	32.207
12	20	00.945
12	20	02.000
12	20	01.230
12	20	01.089

Figure 4.5: Search results, using 4 slave nodes.

that the query that must access the file on 80MB will probably take longer than 1.6 seconds to complete, which is too long when taking into account the recommended time for user interactions. The time consumer is when the partial files must be merged to create the full file. If the cache could hold larger objects the time it takes to retrieve a file would probably go down significantly.

Therefore, the cache should be used when working with files on the cloud.

After reading about how Windows Azure uses blob to store files and taking into account that when there is a file based index this was a natural place for a bottleneck to occur. This is because reading operations from files can be quite costly depending on how they are stored. Therefore, the files should be stored in memory to quickly be able to read them when querying.

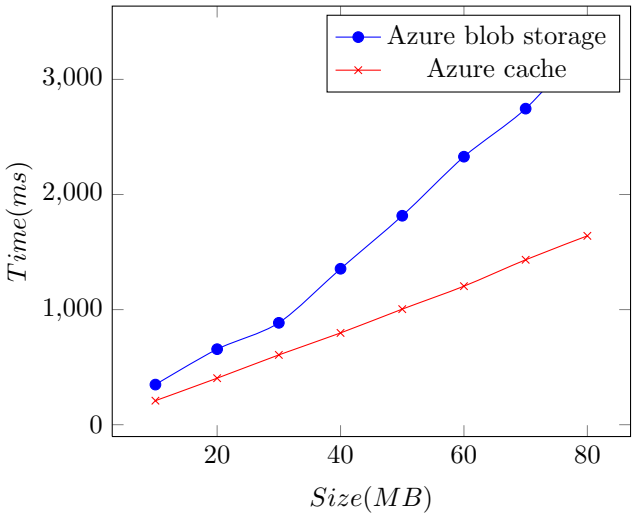

Figure 4.6: Reading performance comparison between Azure blob storage and Azure cache.

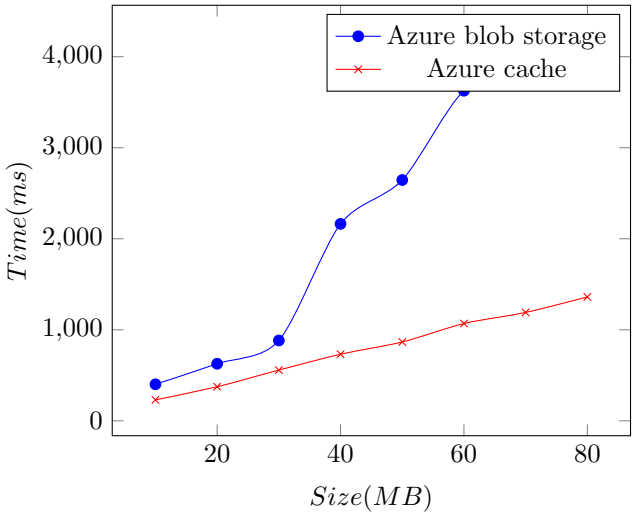

Figure 4.7: Writing performance comparison between Azure blob storage and Azure cache.

Size(MB)	Azure blob storage (ms)		Azure cache (ms)	
	Read	Write	Read	Write
10MB	348.4	401.5	208.7	230.5
20MB	656.2	627.1	404.1	374.4
30MB	885.0	882.6	606.5	557.5
40MB	1354.8	2163.5	797.3	731.5
50MB	1815.4	2645.4	1004.1	866.3
60MB	2328.5	3626.3	1203.5	1068.9
70MB	2745.9	3851.8	1432.8	1190.8
80MB	3325.1	4171.0	1640.7	1360.9

Table 4.6: Comparison of read and write speed of Azure cache with splitting of files larger than 8 MB and Azure blob storage.

4.5.2 Networking results

Since the architecture is a distributed one, the network latency must be taken into account. The test that is done only tests the network latency between two applications in the cloud, and the configuration is the same as how the real system is set up, using web API.

The results in 4.8 show that there is almost no latency when the two applications are working on the same geographic location[4]. The slowest uses only 6 ms for a round trip, whilst the fastest is 1 ms.

Id	Round Trip (ms)
0	6
1	1
2	1
3	6
4	2
5	1
6	1
7	1
8	5
9	3
10	3

Figure 4.8: Communication speed between two applications in the Microsoft Azure cloud.

[4]Northern Europe.

4.6 Discussion

The focus in this thesis has been to find how to process and store big data in the cloud. Where the research questions were going to cover parts of the main question. In this section, each research question will have its own section where they will be discussed.

4.6.1 How can distributed methods be used to process and store big data on the cloud?

The distributed methods and algorithms itself are easily transferable to the cloud environment, except for the methods that utilizes the storage and the assumption that the in memory objects are always retrievable for each call to the system.

This system used in the evaluation based itself on a server-client architecture where the clients had partial indexes. In a distributed environment, when processing big data the clients can each be designated to one computer or virtual computer. In worst case, setting up a computer can be quite time consuming compared to setting up a new client in the cloud environment. When there is a need of more clients, or slaves as they also are called, it only took 5 minutes to publish and start up a new one on the cloud. Each application on the cloud has the possibility to increase the number of instances that will run that application, and the initial thought was to use this when scaling the slave nodes. Interestingly enough, Azure applies automatic load balancing on the instances based on the same role. This means that each instance does not get its own URL, and will therefore not be detected by the master node as separate slave nodes. The solution was to have the slave nodes run as its own application. This solution only has one negative factor, it takes 5 minutes to increase the number of slave nodes instead of 1 minute.

Since the evaluation system had partial indexes distributed amongst the slave nodes, the automatic load balancing was not a very great feature. But, if the system had used slave nodes with whole indexes this would be a greatly appreciated feature. Since the slave nodes would then have been dependent on automatically switch to another node if one became overworked, and the fact that this is a built in feature would reduce the work since you wouldn't have to create your own.

The partial indexing system could also utilize the automatic load balancing by starting up more instances of each slave node. The need for overlapping the files to ensure consistency in the searches would then be rendered redundant. The reason for this is that the overlapping files in the index is designed let the slave nodes go off-line, and the search would not be affected. When the role has two instances the load balancing would take care of the problem that one instance would go off-line. Removing overlapping files would decrease the index sizes, and in turn lead to a smaller amount of slave nodes needed. This is because of the need to have the index files cached in the Azure cache for fast querying. When the files are smaller the cache does not get filled up as fast.

When looking at the system in more detail, you can look at the algorithms taken from the distributed systems. In the evaluation system, the algorithms borrowed and implemented was the merging and the distribution algorithms.

Since the partitioning method was based on local indexes, the merging was done from performing cosine similarity between the results and the query. Comparing the results in 4.3 and 4.5, there were not much difference between searching with a single slave node without any specific merging, or with 4 slave nodes with merging. The search times were approximately the same in both situations. But, the results did not show a search result with thousands of hits. With that many hits it would probably take longer. To make the merging of thousands of results fast the slave nodes could be used together with the map reduce method (see 3.2.1). Each search result could be sent, together with the query to each slave node and in return the cosine similarity could be the returning value. Since the system uses local indexing, and as described in 3.2.3 the local index gets a drop in the precision as the slave nodes increases. Therefore, the merging method should be upgraded to take into account global information so that this drop does not happen.

The distribution algorithm that is implemented was a random distribution algorithm which is described earlier in 3.2.3 and in 3.3.1. As said the algorithm does not take into account file size or semantics in the files. This can make the slave nodes become uneven in sizes and the load could be different from one node to another. But, on the other hand, it is quite fast and not as difficult to implement.

4.6.2 How well does a distributed system in the cloud perform compared to a distributed system that is not in the cloud?

When looking at the results in the evaluation 4.3 and comparing them to the evaluation of the Lucene search engine in 3.1.2. The evaluation done by Middleton and Baeza-Yates proved that Lucene spent 1 hour, 1 minutes and 25 seconds (in total 3685 seconds) to index 2.7 GB (2764.8MB), which is a speed of 0.75 MB/s. Compared to the speed of the evaluation system, 0.23 MB/s, it is more than a 100% faster. But, the evaluation system is not fully optimized. If the evaluation system was fully optimized it would probably be close to that speed.

Comparing the results from indexing with 2MB 4.1 and with 4MB 4.2 file groups, proved that increasing the number of files (higher size) that were sent to the slave node, increased the indexing speed from 0.11 MB/s to 0.23 MB/s. Further increasing the file group would probably increase the speed as well.

One great advantage of the distributed system is that the application can fully utilize the memory on the computer it is running on by keeping index files in memory between calls and removing the need to read from file each time it has to perform a query. When searching there were quite a large difference between the search time in the evaluation by Middleton and Baeza-Yates and the evaluation system. On average the search time for the evaluation system was around 600 ms, whilst the Lucene evaluation was 21 ms. But bear in mind that the document collection for the evaluation system is considerably smaller than the one in the Lucene evaluation. But, as long as the index files were in the Azure Cache and the files were under 8 MB, the search time was not overwhelmingly slow. Querying when reading directly from the blob was, on the other hand, slow and would not

be feasible for an information retrieval system. Therefore, when the first query was sent to the slave nodes (see 4.3 and 4.5), the index files that were going to be used were stored in the azure cache, so that the preceding queries would use the files in the cache rather than from the blob. The performance comparison can be seen in 4.5.1. The cache performance is good, but not as good as it should be when taking into account the recommended time for user interaction is 100 ms. Reading a 80 MB file from the Azure cache took 1.6 seconds, and would itself cause the total query time to overstep the recommended time. A suspicion is that the 8 MB limit to the objects stored in the cache is a restricting factor. Since the files that are over 8MB is split into smaller files, and when the file is needed it is put together by joining these sub-files, a lot of the time would be spent merging the files back together into one larger one.

When it comes to networking latencies between the master and the slaves, the cloud seems to have a tightly integrated system for networking. The networking between the applications, that are located in the same region , are done in great speeds and the latency can be neglected when finding bottlenecks in a system. The results4.5.2 clearly shows that the latency between the two applications that were used for the test was very low. On average, the time it took to call the application and to return with a result took only 2.67 ms. Compared to a distributed system, this is about the same as one would expect from these types of systems.[5]

4.6.3 What challenges are there?

During implementation of the test system, there were quite a few challenges that had to be worked out.

One great problem was how to create and store the index files in something else than a normal file structure without loosing performance. Reading and writing directly with the blob storage made the system very slow. This was quite evident when first creating the system. Initially, the master performed one call to the slave nodes for each file that needed to be indexed. This made the slave node read the index files from the blob to memory for each call. It proved to be very slow, and each file used around 50 seconds to be indexed. How could the index files be kept in memory and then flushed to blob after it was finished reading?

The solution was to send the files in large batches for indexing. From the results 4.1 it showed that now it took 55 seconds to 1 minute 28 seconds to index 2 MB worth of files, and when each file can be from 20KB to a couple of hundred KB, it was a great improvement. This meant that performing the indexing in batches of files it would not have to load the index files from the blob as often, and therefore save that time.

The most optimum solution would be to transfer all the files in one large batch, but there are limitations to this. The JSON deserializer (see 3.3.1) has a maximum object size of 2 GB, assuming that the slave node instance has enough memory to hold that large of an object. There is also the same limitation on objects transferred through the web API. Increasing the object size for the JSON deserializer is not

[5]Running a ping between two computers on the same network usually is around 1-2ms.

a problem, but increasing the size of files and objects uploaded to the web API could pose a problem, since this could be exploited and be used as points of DDOS attacks[6]. The attacker could keep the service busy by uploading many very large files and making the service drop off-line. Therefore, the file batch should be increased to a reasonably large size to optimize the indexing performance.

A different problem that occurred, that also is related to the fact that Azure does not reliably hold objects in memory, is that the index files should be somehow kept in memory for querying, instead of having to load from the blob each time a query is performed. The solution was to load the index files into Azure cache and use it as an in memory storage. The files would then be retrievable between each call, and the azure cache is quite fast. But, as said earlier, the problem with the 8 MB limitation made reading from the cache slower than it should have been. The merging of the index files are done sequentially, so one optimization could be to retrieve the sub files in parallel and then merging them. Then the time to retrieve file from the cache would be the time it took for the slowest thread to retrieve its file and the merging process itself. The best optimization would be that Microsoft removed the size limit altogether.

When it comes to processing big data, there is another great challenge that must be handled. When should the scaling of the slave nodes occur? The evaluation has not been tested on actual big data, but there has been suggestions from the results that shows how to handle big data. The fact that the Azure Cache has a great impact in the querying performance has to be taken into account. Since the cache is limited to a percentage of the total memory on the instance that runs the application, the recommendation should be to increase the number of slave nodes when the index size grows larger than the cache size. Lets say the cache is 50% of a small instance. This would give a cache of 850 MB of memory. If the size of the index is expected to grow larger to 10GB the recommended number of slave nodes should then be, roughly calculated, 12 slave nodes.

4.6.4 Pros and Cons of choosing Windows Azure

When it comes to choosing Windows Azure as a Platform-as-a-Service provider, there has been some ups and some downs.

Windows Azure has had storage in abundance, the networking speeds between applications are great, and the Azure Cache has been great for keeping objects in memory.

The automatic load balancing when increasing the number of application instances is also a great feature for some types of applications, but in exactly this system it was more a pain than a pleasure. As said before, the load balancing would be great if the whole index was distributed to the slave node. The blob storage access point would be the same, and the slave node application could scale up or down depending on the need without having to replicate the index or even perform a new index. The index files would remain untouched, it would not be a need to

[6]Denial of service attack - "Is an attack designed to render a computer or network incapable of providing normal services." (`http://www.w3.org/Security/Faq/wwwsf6.html`)

perform any duplication or any re-indexing. But, if there were one feature could be implemented it would be to allow for choosing when to create a new instance with a new IP or a new instance with the same IP as another instance for automatic load balancing for that instance.

The Azure Cache was not only good to have, there were also some down sides. As said earlier, the maximum object or file size made the search performance decrease, since when the files greater than the 8MB limit had to be split into smaller files so that they didnt overstep the limit. But the most costly operation was when the sub files had to be merged into one large file each time this file had to be retrieved. There are optimizations that can be done, but the cache would in a perfect world be used as the memory in ordinary servers are used. On ordinary servers there is a limitation to the object size, but not anywhere near to the 8MB limitation set in the Azure Cache.

Chapter 5

Conclusion

This thesis started out with one question; How can Big Data be processed and stored in the cloud? This was to be researched by creating a distributed search engine in the cloud, and perform tests and evaluations on it. To find the search engine the cloud platform must have been chosen first, by setting up a few criteria that the cloud should be able to cover.

The different cloud platforms should be able handle storage, it should be able to handle fast processing of search queries and it should have a simulated runtime environment. Since all of the cloud platforms had good numbers on most of these areas, the limiting factor was how much the free trial would give you, and therefore would be used in this thesis. The platform that prevailed was Microsoft's Windows Azure, since the free trail gave the most storage, and most of the specifications were slightly higher than the other platforms. It also has a runtime environment that was important to set up the projects, for testing on the cloud.

When the cloud platform was chosen, the next step was to choose which search engine to use. Since Windows Azure only support .NET languages, such as C#, C++ (Managed), Visual Basic, etc. But to fully take advantage of the features in Windows Azure C# or visual basic were the preferred languages, since they are easier to develop web projects and utilize the full .NET library. Looking at

The choice of search engines fell on the veteran of the open source search engines, Lucene. To be more precise it was the C# version of the original Java written version, Lucene.Net. One of main reasons for the choice was that Lucene is open source, which means that the code is open and can be rewritten to suit your needs. Also, it is thoroughly evaluated and compared to other search engines, and it was one of the better search engines. Moreover, it has been around for many years and there has been many search engines that have been built, based on it. Therefore, there are probably many problems circulating, that already has been solved and can help progressing the project faster forward.

The search engine was built, using the distributed method of partial indexes that were stored in Windows Blob Storage. The indexes were locally indexed, which means each slave node (client) did not know of the others and did not have any global term information. The files were distributed amongst the slave nodes

using a random distribution algorithm, which meant that there were a possibility of an uneven distribution, but for evaluation purposes was enough.

When the master node queried the slaves, they performed the search and returned the hits together with information that allowed to perform a cosine ranking on the master. When the initial queries were performed, the index files were loaded from Windows Blob Storage to the Azure Cache, where the cache worked as memory so that each call to the slave would have the index files available. The files had to be loaded into the cache, since Windows Azure is stateless and two calls to the web API could in theory go to two different servers running the same application.

The results showed that when the files were indexed the speed of the throughput of the indexing increased together with the increase of file batches that were sent to indexing. Which showed that each time a new batch were sent to the slave nodes for indexing, the index files had to be loaded from the Blob Storage. Loading from the Blob Storage proved to be quite a time consuming operation,and making the file batches larger in turn decreased the number of loadings that had to be performed. Since indexing big data is quite time and resource consuming, it is essential that the performance of this operation is optimum.

The search evaluation showed that the performance in this operation also suffered from a delay when the index files had to be loaded from the blob. The first query to the slave nodes, made the index files load from the blob to the cache, therefore this operation could take an unreasonable amount of time. Nevertheless, the preceding queries which used the index files from the cache to perform the query proved to be quite faster. Finding this lead to an evaluation that compared the reading and writing operation using the blob storage and the cache, where the cache was clearly faster. The only problem with the cache was that it could only hold objects or files that were of the size of 8 MB, which meant that the objects that were transfered to and from the cache had to be split and merged to and from smaller files.

As the slave nodes' indexes grows larger it showed that the Azure Cache is so important for the query performance that it should always have all the index files loaded at all times. Therefore, scaling the slave nodes up or down should be decided by the space left in the cache.

When it comes to processing and storing big data in the cloud, it is very possible. Methods designed for distributed systems are mostly transferable to a distributed system in the cloud. But, there are performance issues with the storage that needs to be recognized and resolved. Comparing a distributed system in the cloud and one not in the cloud, the results concluded that, with some more tweaking they might be equally fast. Moreover, when scaling the system, the cloud system is much easier to scale.

5.1 Further Work

To take this work further, the system should be optimized to perform as fast as possible, and then it should be tested using big data. The system should be evaluated to see if the indexing can handle the large amount of data and if the

query still is able to work fast. One optimization that can be done is reading and writing to and from the data cache in parallel to make splitting and merging the index files faster, which in turn will make the queries faster.

Another evaluation the system should undergo is a stress test. When there is big data, there is usually a lot of activity on the search engine, and to see if it can handle the requests a stress test should be done.

The possibility of creating instances with same IP and using the automatic load balancing is a very interesting area. The slave nodes should increase the number of instances it is per IP and see how the automatic load balancing affects the system. Will it be able to handle more requests? Will it be possible to neglect overlapping files and still be able to take one slave node off-line without interruptions? These are some very interesting questions that should be answered.

Appendix A

Results

A.1 1 slave node without scaling

Indexing in file groups of 2MB.

Group	Time (min:sec)	Size (MB)
0	01:25	3.40
1	00:59	13.61
2	00:55	20.48
3	01:05	27.12
4	00:58	33.83
5	00:56	40.40
6	01:02	47.34
7	01:06	54.12
8	01:46	60.94
9	02:19	33.11
10	01:10	72.54
11	01:09	79.12
12	01:13	85.74
13	01:28	92.30
14	01:25	98.73

Indexing in file groups of 4MB.

Group	Time (mm:ss)	Size(MB)
0	00:52	6.76
1	00:47	26.56
2	01:08	40.01
3	00:59	53.52
4	01:12	66.94

Group	Time (mm:ss)	Size(MB)
5	01:02	79.91
6	00:59	92.68
7	01:09	105.58
8	01:12	118.84
10	01:08	141.98
11	01:14	155.18
12	01:31	168.72
13	01:32	182.06
14	01:28	195.03

Searching with 1 slave node.

Group	Hits	Time (ms)
0	4	06.661
0	4	00.256
0	4	00.232
0	4	00.720
0	4	00.143
1	14	06.244
1	14	00.672
1	14	00.486
1	14	00.526
1	14	00.680
2	20	07.547
2	20	00.651
2	20	00.533
2	20	00.702
2	20	00.514
3	20	08.788
3	20	00.463
3	20	00.460
3	20	00.667
3	20	00.434
4	20	10.083
4	20	00.514
4	20	00.304
4	20	00.518
4	20	00.379
5	20	16.777
5	20	00.455
5	20	00.509
5	20	00.458
5	20	00.394
6	20	12.665

Group	Hits	Time (ms)
6	20	00.466
6	20	00.714
6	20	00.590
6	20	00.434
7	20	14.511
7	20	29.155
7	20	00.430
7	20	00.275
7	20	00.431
8	20	27.829
8	20	00.521
8	20	00.845
8	20	00.467
8	20	00.463
9	20	18.341
9	20	00.510
9	20	00.385
9	20	00.569
9	20	00.394
10	20	15.709
10	20	00.540
10	20	00.304
10	20	00.643
10	20	00.338
11	20	17.041
11	20	00.351
11	20	00.334
11	20	00.325
11	20	00.406
12	20	18.854
12	20	00.323
12	20	00.442
12	20	00.217
12	20	00.263
13	20	20.241
13	20	00.332
13	20	00.270
13	20	00.244
13	20	00.275

A.2 2 slave node without scaling

Searching with 1 slave node.

Group	Time (mm:ss)	Size(MB)
0	00:59	3.44
1	00:55	17.25
2	00:59	38.02
3	01:04	65.54
4	00:59	99.87
5	01:01	140.85
6	01:19	188.87
7	01:42	243.77
8	01:41	305.58
9	03:03	356.70
10	02:02	429.94
11	01:59	509.83
12	01:50	596.40
13	01:59	689.59

Searching with 2 slave nodes.

Group	Hits	Time (ms)
0	4	04.198
0	4	00.097
0	4	00.223
0	4	00.146
0	4	00.107
1	14	04.364
1	14	00.300
1	14	00.632
1	14	00.294
1	14	00.374
2	20	07.358
2	20	00.423
2	20	00.464
2	20	00.623
2	20	00.715
3	20	07.840
3	20	00.530
3	20	00.351
3	20	00.503
3	20	00.515
4	20	10.313
4	20	00.620
4	20	01.151
4	20	00.870
4	20	00.769
5	20	12.474

Group	Hits	Time (ms)
5	20	00.751
5	20	00.431
5	20	00.692
5	20	00.476
6	20	21.591
6	20	00.657
6	20	00.779
6	20	00.471
6	20	00.685
7	20	21.195
7	20	00.567
7	20	00.447
7	20	00.673
7	20	00.567
8	20	22.685
8	20	00.448
8	20	00.483
8	20	00.467
8	20	00.555
9	20	23.025
9	20	00.779
9	20	00.519
9	20	00.455
9	20	00.679
10	20	17.273
10	20	00.479
10	20	00.484
10	20	00.764
10	20	00.904
11	20	18.161
11	20	00.464
11	20	00.508
11	20	00.558
11	20	00.562
12	20	21.751
12	20	00.597
12	20	00.484
12	20	00.732
12	20	00.689
13	20	26.000
13	20	00.704
13	20	00.822
13	20	00.602
13	20	00.934

A.3 3 slave node without scaling

Indexing with 3 slave node using file groups of 2MB.

Group	Time (mm:ss)	Size(MB)
0	01:47	6.842406273
1	01:34	27.43525982
2	01:37	41.28302288
3	01:34	54.67120934
4	01:59	68.19398308
5	02:02	81.42880535
6	02:49	95.42016029
7	02:15	109.0745096
8	02:27	122.8222675
9	02:32	133.9340506
10	02:36	145.8539791
11	02:53	159.0913811
12	02:35	172.3967342
13	02:34	185.5799227
14	02:15	198.5223694

Searching with 3 slave nodes.

Group	Hits	Time (ms)
0	4	06.963
0	4	00.223
0	4	00.245
0	4	00.147
0	4	00.162
1	14	08.094
1	14	01.187
1	14	00.776
1	14	00.721
1	14	00.472
2	20	10.393
2	20	00.896
2	20	00.614
2	20	01.076
2	20	00.802
3	20	12.585
3	20	00.740
3	20	00.972
3	20	00.803
3	20	01.233

Group	Hits	Time (ms)
4	20	14.773
4	20	00.865
4	20	00.949
4	20	00.948
4	20	01.087
5	20	29.656
5	20	01.114
5	20	01.505
5	20	01.727
5	20	01.633
6	20	25.999
6	20	00.722
6	20	05.499
6	20	00.714
6	20	00.968
7	20	28.007
7	20	00.724
7	20	00.695
7	20	00.715
7	20	00.756
8	20	29.037
8	20	00.716
8	20	01.029
8	20	00.664
8	20	00.734
9	20	21.946
9	20	01.060
9	20	00.472
9	20	01.197
9	20	00.525
10	20	22.036
10	20	00.698
10	20	00.922
10	20	00.707
10	20	00.787
11	20	28.783
11	20	01.013
11	20	01.089
11	20	00.857
11	20	00.752
12	20	24.685
12	20	00.891
12	20	00.554
12	20	00.620

Group	Hits	Time (ms)
12	20	00.690

A.4 4 slave nodes without scaling

Indexing with 4 slave node using file groups of 2MB.

Group	Time (mm:ss)	Size(MB)
0	01:58	6.88
1	01:35	27.60
2	01:34	41.54
3	01:49	55.02
4	01:38	68.64
5	01:54	81.96
6	03:19	96.03
7	02:18	109.77
8	02:04	123.60
9	02:23	134.58
10	02:14	146.82
11	02:08	160.41
12	02:54	174.05

Searching with 4 slave nodes.

Group	Hits	Time (ms)
0	4	06.906
0	4	00.265
0	4	00.174
0	4	00.180
0	4	00.151
1	14	07.049
1	14	00.481
1	14	00.560
1	14	00.799
1	14	00.420
2	20	09.032
2	20	00.761
2	20	00.826
2	20	00.547
2	20	00.625
3	20	14.839
3	20	01.321
3	20	00.954
3	20	01.102

Group	Hits	Time (ms)
3	20	01.025
4	20	15.467
4	20	01.149
4	20	01.290
4	20	01.201
4	20	00.795
5	20	43.775
5	20	01.534
5	20	01.645
5	20	01.847
5	20	01.780
6	20	20.876
6	20	00.811
6	20	00.896
6	20	00.867
6	20	00.841
7	20	20.852
7	20	00.938
7	20	00.800
7	20	00.596
7	20	00.920
8	20	26.640
8	20	00.636
8	20	00.980
8	20	00.690
8	20	01.089
9	20	24.067
9	20	01.134
9	20	01.005
9	20	01.146
9	20	00.908
10	20	18.670
10	20	00.855
10	20	00.624
10	20	01.024
10	20	00.983
11	20	23.047
11	20	00.832
11	20	01.187
11	20	00.788
11	20	00.902
12	20	32.207
12	20	00.945
12	20	02.000

Group	Hits	Time (ms)
12	20	01.230
12	20	01.089

Bibliography

[1] 101tec Inc (2010). How katta works. http://katta.sourceforge.net/documentation/how-katta-works.

[2] Agrawal, D., Das, S., and Abbadi, A. E. (2011). Big data and cloud computing: current state and future opportunities. In Proceedings of the 14th International Conference on Extending Database Technology, EDBT/ICDT '11, pages 530–533, New York, NY, USA. ACM.

[3] Baeza-Yates, R. and Ribeiro-Neto, B. (1999). Modern Information Retrieval. Pearson Education Limited.

[4] Cambazoglu, B. B. and Aykanat, C. (2006). Effect of inverted index partitioning schemes on performance of query processing in parallel text retrieval systems .

[5] Chapell, D. (2008). A short introduction to cloud platforms: An enterprise-oriented view. http://www.davidchappell.com/CloudPlatforms--Chappell.pdf.

[6] clouds360.com (2013). The top 20 platform as a service (paas) vendors. http://www.clouds360.com/paas.php.

[Corp.] Corp., M. Pricing details.

[8] Corp., M. (2012a). About windows azure caching. http://msdn.microsoft.com/en-us/library/windowsazure/hh914161.aspx.

[9] Corp., M. (2012b). Common language runtime (clr). http://msdn.microsoft.com/en-us/library/8bs2ecf4.aspx.

[10] Corp., M. (2012c). Understanding block blobs and page blobs. http://msdn.microsoft.com/en-us/library/windowsazure/ee691964.aspx.

[11] Corp., M. (2012d). Understanding quotas for windows azure shared caching.

[12] Corp., M. (2013a). Asp.net mvc overview. http://msdn.microsoft.com/en-us/library/dd381412(VS.98).aspx.

[13] Corp., M. (2013b). How to use the windows azure blob storage service in .net. http://www.windowsazure.com/en-us/develop/net/how-to-guides/blob-storage.

[14] Corp., M. (2013c). How to use windows azure caching. http://www.windowsazure.com/en-us/develop/net/how-to-guides/cache/.

[15] Corp., M. (2013d). Service level agreements. http://www.windowsazure.com/en-us/support/legal/sla/.

[16] Corp., M. (2013e). Web api. http://www.asp.net/web-api.

[17] Corp., M. (2013f). What is windows communication foundation. http://msdn.microsoft.com/en-us/library/ms731082.aspx.

[18] Corp., M. (2013g). Windows azure. http://www.windowsazure.com.

[19] Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplifed data processing on large clusters. Magazine Communications of the ACM - 50th anniversary issue: 1958 - 2008.

[20] Engineering, F. (2012). Under the hood: Scheduling mapreduce jobs more efficiently with corona. Technical report, Facebook.

[21] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., p. Leach, and Berners-Lee, T. (1999). Hypertext transfer protocol – http/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[22] Foundation, T. A. S. (2013). What is apache hadoop? http://hadoop.apache.org.

[23] Inc., G. (2012a). Blobstore java api overview. https://developers.google.com/appengine/docs/java/blobstore/overview.

[24] Inc., G. (2012b). Datastore overview. https://developers.google.com/appengine/docs/java/datastore/overview.

[25] Inc., G. (2012c). Google app engine. https://developers.google.com/appengine/.

[26] Inc., G. (2012d). The google cloud storage api. https://developers.google.com/appengine/docs/java/googlestorage/.

[27] Inc., G. (2012e). Mapreduce overview. https://developers.google.com/appengine/docs/python/dataprocessing/overview.

[28] Inc., G. (2013). Memcache java api overview. https://developers.google.com/appengine/docs/java/memcache/overview.

[29] Jacobs, A. (2009). The pathologies of big data. Commun. ACM, 52(8):36–44.

[30] Manning, C. D., Raghavan, P., and Schtze, H. (2008). Introduction to Information Retrieval. Cambridge University Press.

[31] Middleton, C. and Baeza-Yates, R. (2007). A comparison of open source search engines.

[32] Miller, M. (2013). Solrcloud. `http://wiki.apache.org/solr/SolrCloud`.

[33] Nielsen, J. (1993). Usability Engineering. Morgan Kaufmann.

[34] Ribeiro-Neto, B. A. and Barbosa, R. A. (1998). Query performance for tightly coupled distributed digital libraries. In Proceedings of the third ACM conference on Digital libraries, pages 182–190, New York, NY, USA. ACM.

[35] Schatz, M., Langmead, B., and Salzberg, S. (2010). Cloud computing and the dna data race. Nat Biotechnol.

[36] Services, A. W. (2013). What is the aws free usage tier? `http://docs.aws.amazon.com/gettingstarted/latest/awsgsg-freetier/TestDriveFreeTier.html`.

[37] Varia, J. and Mathew, S. (2012). Overview of amazon web services.

[38] Voorsluys, W., Broberg, J., and Buyya, R. (2011). Introduction to cloud computing.

[39] Zhang, Q., Cheng, L., and Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. Journal of Internet Services and Applications.