Bachelor Thesis
LIACS, University of Leiden

The Anatomy of a Collaborative Image Search System

Author:
Menno Luiten (s0345296)

Supervisor:
dr. Mark Huiskes

Abstract

"An image tells more than a thousand words". Yet, while text searching on the Internet has become indispensable, with fast and scalable techniques, searching for *images* has not. The purpose of this paper is to design an infrastructure for an Internet-based Collaborative Image Search System (hereafter also called CISS), using existing Information Retrieval techniques and adjusting them to work play nice with multimedia. For this purpose I will use relevance feedback, distributed indexing and distributed computing using Hadoop and HBase.

1 Introduction

Image analysis and other techniques for describing multimedia are becoming more advanced and accurate, while more techniques for information retrieval systems are getting more accessible. While many multimedia information systems are based on KD-trees[1], I will try to combine several branches of research in a relatively efficient way in designing a new search engine based on Aspect-based Relevance Learning[2] (hereafter called ARL), using techniques also commonly used by free text information systems. Also we would like to create collaboration among users of this search engine by using collaborative filtering techniques, thereby creating a Collaborative Image Search System (CISS). The techniques will be implemented in a system named CISSLE[1]. We assign aspects to the images using several types of analysis. This has several implications: the image analysis has to run fast, reliable and accurate to keep up with the immense amount of data on the web, while the search system itself should hold large amounts of data that is not purely based on text.

Because this is (only) a Bachelor Thesis, it would be unrealistic to aim for a fully functional search system with all features implemented. So many of the actual features will be discussed but not implemented. In this report

[1] http://www.cissle.com

I will first introduce some general background and anatomy of search engines. I will then introduce the techniques used with CISSLE, and specify the specific design choices made during the development of CISSLE. I will mostly describe the the most important aspects of this project: relevance feedback and distributed computing using MapReduce. Then some results of CISSLE will be discussed and I will conclude.

2 Anatomy

The anatomy of CISSLE will have much in common with any other document-based search system. In this section I will explain the general structure of such a search system and explain in what ways my proposed search system is different and why. Later, in Section 4 I will specify how I implemented these techniques in CISSLE. I will especially use techniques that provide massive data storage and provide a simple way to scale *horizontally.* That is; adding servers rather than upgrading servers. The reason is that the costs of scaling horizontally are linear, while the costs of scaling vertically are often exponential. For example, if we need to double the performance of a 10 server cluster, adding 10 servers will cost exactly the same as the initial investment. Upgrading the existing servers to double the performance will however mean that all processors, memory, networking equipment etc must be upgraded. At first this might be affordable, but there's a limit; the current technological advancements. If all servers are equipped with 4Ghz quad core processors, the costs of doubling processors to 8GHz quad cores or 4GHz hexacores is very expensive or even not possible. Many search systems therefore use parallel techniques that are optimized to scale well horizontally. I will expand on that subject in Section 3. First, I will quickly introduce the general subsystems of our search system.

2.1 Crawler

A crawler provides the search system with the data that needs to be indexed. In our case this crawler should search the Internet for images and store it together with some metadata (e.g., some sort of PageRank[3]) and some useful context-based information (such as user-created annotations and copyright information). The crawler stores this information in what is often called a *repository.*

2.2 Analysis

The analysis is a bit more complicated than the use 'scan the document for keywords' that text search engines can afford. Each image in the *repository* has to be analyzed by some algorithm(s) which output *aspects* of the provided image. I shall call this annotating the images. I will try to implement this in a way that algorithms can be added and modified, since this is a research field that is still heavy in development. As long as the aspects lead to useful results and can be represented textually, it does not matter what the aspect represents. We store the list of aspects per image as a vector, which we will call the *image index*.

2.3 Indexing

Indices in Information Retrieval are basically data structures to rapidly identify documents by their contents. In search system the most common index is the *inverted index*. The general idea of this index is to list per aspect, the documents or images that match said aspect. So after analyzing the image, we have an *image index* linking each image to one or more aspects. For the inverted index, we invert this data by listing for each aspect, the images that contain that aspect. We call the list of images for an aspect, a *posting list*.

2.4 Queries

While the previous subsystems are invisible for the end-user, this changes when we come to *queries*. Queries are the way the user interacts with the search system. The most successful search engines (Yahoo!, Google, Bing) have only one query field, which searches one or multiple indices for matches. I will replicate this simple interface, and let the query search for matching aspects. A notable challenge is to provide enough annotations that these initial searches provide plenty results. However, in CISSLE, the query field is not the main feature. We will use a drag&drop *example set* to refine image results using relevance feedback (RF). This causes the queries to be more complicated, but the magic trait of search engines is that the seemingly simple queries have enormous parallelism and can therefore be executed very fast. So our queries will invoke additional sub-queries depending on the example set, hopefully without suffering too much in terms of performance.

2.5 Relevance Feedback

Relevance feedback is a method of letting a user define whether or not a match is relevant, given a query *result set*. In the type of explicit relevance feedback we will use, the user can select images matching certain aspects (such as color, subject, etc) into an *example set*. With the data from the example set, the query can be refined (e.g., by adding the top 10 terms from each selected match to the search query), thus finding images matching aspects from the original query ánd the example set.

2.6 Collaborative Filtering

Collaborative filtering is a process that filters datasets based on patterns executed by multiple agents, in our case multiple queries. Certain patterns in the dataset might indicate that the weight of some term on a certain image should be higher or lower than their current weight. We perform this filtering based on data collected from the relevance feedback, as well as click data. If none of the users searching for 'car' are content with some image X, we might want to adjust the weight of the aspect 'car' on that image X downwards, to make it less relevant for future requests. This is a research field in itself and many large webshops use similar algorithms to recommend products to their customers.

3 Distributed Computing

I started my literature research by looking at the major search engine players, such as Google, Yahoo! and Bing. Unfortunately, since search engines can be very profitable, much of the inner workings of these search engines are well-kept secrets. However some general information can be extracted from several papers and articles. I focused mainly on the book by (Information Retrieval) and on early Google papers like (referenties), which should outline the general structure of a proper search engine.

All major search engines have one thing in common: distributed computing and indexing. Not only because of the sheer size the dataset, which obviously is too big to fit on one computer, but also because of the potentially huge amount of queries on the database. Multimedia Information Retrieval systems often involve KD-trees as can be read in [1]. I would like to find out wether the techniques used in 'regular' search engines, can be useful on multimedia datasets.

In the early stages of my research I tried to expand my current knowledge on relational databases and web-based application development to encompass the field of large-scale databases and low-latency queries. Using this preliminary research I designed a service infrastructure as depicted in Figure **??**. After some brief experiments using a single MySQL[4] database, it seemed there was no linear scaling and storage was quite limited. I stumbled upon several articles [5, 6] regarding Database Sharding, often using MySQL and PHP for large-scale PHP sites. In short, sharding is a very application-dependent technique to distribute tables over multiple database servers, thereby placing data for a certain user on server1, while placing data for another user on server2. However, as I was reading the comments on these articles, I found many authors criticizing that this sharding requires a lot of manual bookkeeping (e.g., "on which server is data for user2 stored?"

and “what is we want to move user2 to server3?”) and is actually just an (bad) emulation of actual distributed computing solutions such as parallel databases or MapReduce. I decided to research one of these ‘real’ solutions for CISSLE.

So I started my search for a distributed computing and indexing solution where almost every human being on the planet starts of their Internet search: at Google®. Google’s original paper is called *‘The Anatomy of a Large-Scale Hypertextual Web Search Engine’*[3], and later introduced a concept called MapReduce[7]. The Google architecture is build around a cluster of commodity machines, instead of a heavy-duty supercomputer. These relatively cheap and often not uniform servers gave an easy solution using massive parallelism for certain problems. Even later, Google introduced Bigtable[8]: a distributed, persistent row-value data store, which can also be used as input and/or output of MapReduce jobs. Because of this, the combination of MapReduce and Bigtable can be used to handle large amounts of data, which in turn can be processed in parallel on a cluster of arbitrary size. It turns out that this can be used particularly effectively in a search system for the indexing process.

3.1 MapReduce

MapReduce[7] is essentially a two-phase algorithm, of which each part of each phase is executed in parallel and possibly distributed over multiple servers. The first phase is called the *map* phase, executed by a *Mapper*. Each Mapper retrieves a subset of the total dataset from the (often very large) input, and outputs one or multiple <key, value>pairs, which will be used as input for the *reduce* phase. The *Reducer* receives **all** values pair of a certain key. In the original paper this is displayed as

```
map     (k1, v1) → list(k2, v2)
reduce  (k2, list(v2)) → list(v3)
```

For example, if we want to count all the words in a (very large) document, the Mapper will receive a part of the document as input, and output <word, 1> for every word it encounters. The Reducer then retrieves a list of 1’s (one for every occurrence of the word). It will output <word, sizeof(values)>, thereby returning the number of times the word occurs in the document. Because of the high parallelism that can be achieved by both the Mapper and the Reducer, this algorithm can be executed very efficiently using a MapReduce cluster.

In figure 1 a more detailed overview of the Map/Reduce algorithm is shown and pseudocode for the word-counting algorithm is shown in Appendix A.

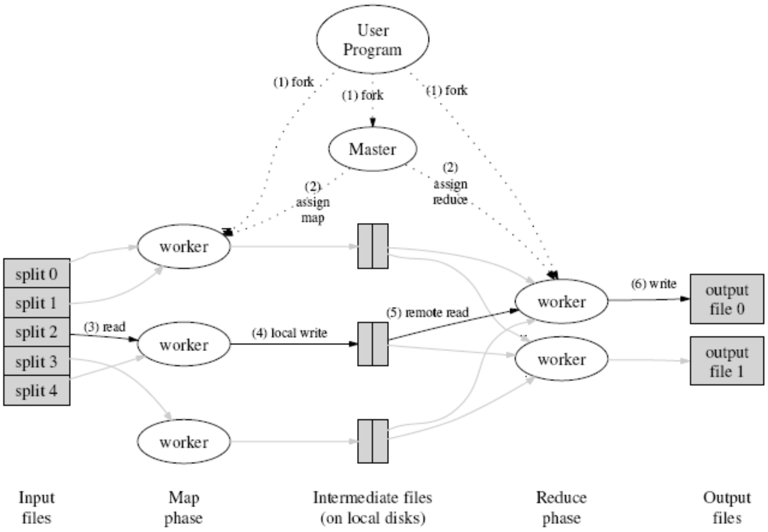


Figure 1: Map/Reduce

DeWitt and Stonebraker[9], pioneers in parallel databases and shared-nothing architectures, have made comments about MapReduce being a step backwards rather then a progression in terms of databases, mainly because of its lack of relations. They mention parallel databases as a proper alternative. In my eyes, they miss out on the fact that MapReduce is not (and does not aim to be) a replacement for relational databases, but rather a paradigm to provide easy parallism. I found MapReduce flexible enough for all my goals (as will be elaborated upon in Section 4) and found no proper affordable parallel databases, let alone open-source ones. Besides, if Google can afford to use it, I certainly can.

3.2 Hadoop

Hadoop is an open-source implementation of the MapReduce[7] paradigm and has gathered an active user base and many related projects. Major users include major search engine players like Yahoo![10], Facebook, Rackspace[11] and Microsoft (who acquired the company that designed Hadoop: Powerset)[12]. With Hadoop, it is relatively easy to configure a cluster with a distributed filesystem (HDFS) and coordination for MapReduce jobs. While Hadoop itself is a MapReduce environment, many contributed projects have expanded its application domain:

- Zookeeper[13] is a distributed lock manager with automatic fail-over abilities. It somewhat resembles Google's Chubby[14].

- HBase[15] is a persistent, distributed row store build based on Google's Bigtable. Recent version of HBase use Zookeeper for load-balancing and fail-over.
- Mahout[16] can execute machine learning libraries on a Hadoop cluster.
- Pig[17] is a high-level language for programming a MapReduce cluster, similar to Sawzall used by Google.

A combination of these projects can provide us with a flexible tool set for our project. While analyzing images is not necessarily a strong point of Hadoop, it can make sense to use Hadoop for this purpose, since it will provide major advantages in terms of indexing and collaborative filtering in a later stage.

3.3 HBase

Our main focus with the related Hadoop projects will be on HBase. HBase is a distributed, column-oriented data store modeled after Google's Bigtable[8]. In the Bigtable paper, Chang et al. explain they use this data store for large data sets such as Google Maps, ... and even parts of their search engine. HBase is designed to deal with Hadoop's inability to efficiently work with small files and/or data. But mainly because of the parallelism and database-like structure, I decided to use it to store all my *indexes* and *repositories*.

The basic architecture of HBase consists of one *Zookeeper node*, one *master node*, and one or multiple *region servers*. One server can run multiple nodes (e.g., one server can run both the master and a region server, although this would not be very safe). The Zookeeper node takes care of master node fail-over, the master node coordinates all actions of and between the region servers, while the region servers actually contain data and serve requests. The master knows where all the parts (or 'splits') of the tables are, but after requesting the location of a split of the table, the client can communicate directly with the region server. These parallel properties make it very scalable and fast.

3.4 Mahout

Another interesting project is Apache Mahout. While the project isn't far in development, Taste[16] can be very useful for our collaborative filtering.

4 CISSLE

In this section I will elaborate on what my specific ideas for CISSLE are, and how I have implemented (parts of) it.

4.1 Main focus

After researching the possibilities and figuring out common techniques in Information Retrieval, I tried to focus my efforts on very general parts of the search engine for this project. Since time was rather limited, I could not afford to write a complete version of CISSLE from the ground up. My intended end-goal was to have a **functional search engine interface** using **drag and drop relevance feedback**, and executing **queries using ARL** on **HBase indexes** on a **Hadoop cluster**. Ideas for other aspects of the search system have sprung to mind and will be discussed in this section, but I will elaborate mostly on the subsystems that I have implemented.

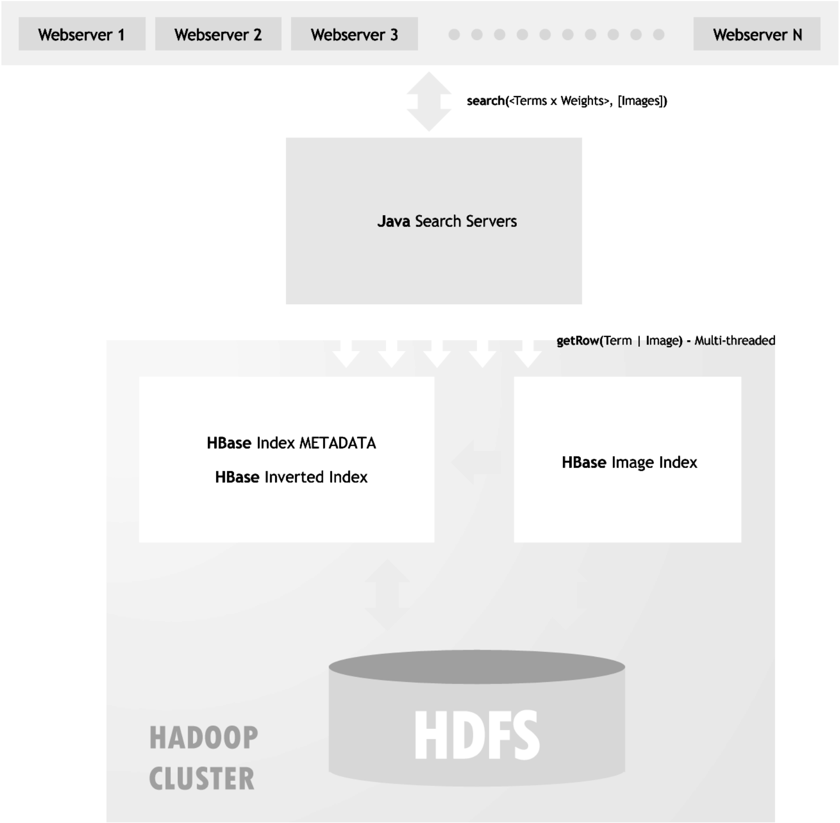


Figure 2: Basic server architecture of CISSLE

4.2 Data set

The first concession made, was scrapping the crawler. Writing our own crawler or writing a plug-in should not be an immense task, but Mark is one of the maintainers of the MIRFLICKR25000 data set[18] and Aspect

Explorer. The MIRFLICKR collection consists of 25,000 images gathered from Flickr[19], including licenses and tags. Aspect Explorer is a program that demonstrates ARL[2] and also calculates aspects on the image set. This is a very feasible test-set for my online search system. Ideally, the results from CISSLE and Aspect Explorer should be identical. I used aspect calculations made by Aspect Explorer and transformed these into a HBase index, instead of letting crawlers collect data from the Internet and analyzing them myself. This should, however, not be a large task to complete. In future work I would like to write a plug-in and output writer for the Heritrix or Nutch crawler, that collects images with metadata and writes output in a HBase compatible format. These two crawler projects provide a good part of the crawling process, also taking care of proper connection management, robots.txt in- and exclusions, duplicate visits, depth limits, etc.

4.3 A short word on ARL

ARL, or *Aspect-based Relevance Learning*, is a technique to test which aspects in a query are relevant to a user, given some user relevance feedback and information about the collection. In our case this technique enables us to see which aspects are important based on the images in the example set, and also rank the images resulting from the aspect relevance, resulting in an ordered result set. For more information on ARL I recommend reading [2]. The original ARL as described is implemented by using a bit matrix (images as rows, aspects as columns), in which 0 means 'image does not contain aspect' and 1 means 'image does contain aspect'. To facilitate collaborative filtering (and thereby gradually decrease the importance of certain aspect on a certain image), I replaced the bits by a floating point value.

4.4 Indexing

With a HBase index populated with aspects on the MIRFLICKR collection, the next step is the indexing of the aspects, thereby creating an inverted index. For this purpose I wrote a Hadoop MapReduce program featuring weighting and p_{db} calculation. The map-phase of the program has an input of imageID:aspectID:weight, and outputs aspectID:imageID:weight. The reduce phase, receiving all imageID's per aspectID, outputs aspectID: list<imageID:weights> into a HBase table. Also it calculates the p_{db} for every aspect, by dividing the size of the list of imageID's by 25,000 (the size of the collection).

We can use the weight given to the aspects by the indexing to divide this index into multiple tiers. This will potentially decrease the size of the posting lists and thereby the result set that needs to be sent over the network, intersected and sorted. Weights can be changed through the collaborative filtering techniques, and therefore images can travel from one tier to another. Let's take, for example, 4 tiers:

1. Weight 1.0 through 0.9 − *Top Tier / Champion list*
2. Weight 0.9 through 0.75
3. Weight 0.75 to 0.5
4. Weight lower than 0.5 − *Bottom Tier*

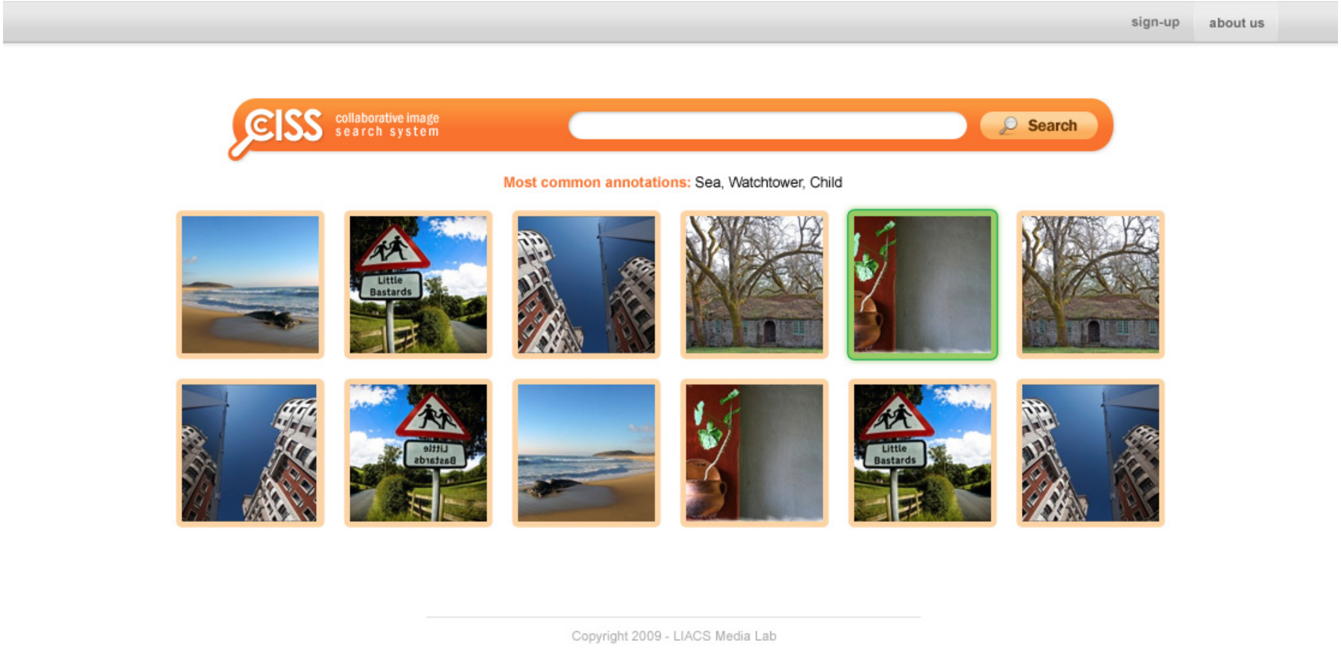
When searching for 1 feature, this will always return the best results in the top tier. Also, this small top-list can be kept in-memory for optimal performance, since it will contain very often accessed data. However, when searching for multiple terms, it might be possible that the intersection of multiple posting lists returns very few results (e.g., if all images matching 'cat' and 'dog' result in very high p-values). In this case we need to retrieve the results from lower tier(s), which may or may not be in cache. Digging through lower tiers keeps getting more expensive and less likely to be contained in cache. So it makes sense to limit the depth of the search.

4.5 Web interface

For the first part of the engine I focused my attention on javascript libraries capable of doing drag and drop. I had some experience using Dojo[20], and I found a very useful tutorial on drag and drop[21]. I adjusted it to feature only a example set container. I also adjusted the search button to append the file names/ID's of the images in the example set container. The request is sent through AJAX to a php script that propagates the request to a Search Server. When all results are gathered from HBase, the request is returned and an ordered/ranked list of images is sent to the browser. These are presentes in a matrix-like table and the user is allowed to use these in the drag and drop container to use the image relevance feedback feature.

4.6 Search server

The search server communicates between the web interface and the indexes stored in HBase, and is responsible for sorting and relevance ranking. Together with research and setting up the Hadoop and HBase cluster, this is what I spent most of my development time on. I started out with a simple version that looked up rows on a specified table and let the PHP script determine the ranking. Later versions were programmed in Java and each request from the webserver is handled by a Worker Thread, which in turn divides query terms into "buckets". Each bucket contains terms that should



Most common annotations: Sea, Watchtower, Child

Copyright 2009 - LIACS Media Lab

Figure 3: Preview of CISSLE interface

be located on one region server. Then, each bucket is given it's own thread, and executes a search on the region server. The buckets then return asynchronously and results are intersected. This setup allows multiple clients to connect without much latency. Also, because each client has multiple simultaneous requests to the region servers, they are gathered faster than a single query. Thus increasing the number of region servers should improve the query times and decrease the load per region server.

When this version proved that acceptable lookup results could be achieved, I started developing a version that could handle Aspect Relevance Learning (ARL). The current version has three distinct phases that require HBase requests. First, it looks up aspects of the images in the query string and example set, then looks up p_{db}-values of those aspects and calculates the ARL ranking and selects relevant aspects. Thirdly, for all relevant aspects it retrieves the posting lists and combines them with some ranking (using p-values) into a final result set. The p-values are calculated with

$$p(N) = \sum_{i=N}^{n} \binom{n}{i} p_{db}^{i} (1 - p_{db})^{(n-i)}, \tag{1}$$

where n is the current total number of positive examples, and N the number of positive examples that possess an aspect. The p-value cutoff is defined by the function

$$p_{min} = 1/(10 + N^5). \tag{2}$$

5 Experiments

As a testing environment I first set up a Amazon EC2 cluster running Hadoop 0.19.0 and HBase 0.19.0. EC2 was a logical choise because of its per-instance-hour billing, allowing changes in cluster size without heavy investments. Also, EC2 scripts for HBase 0.19 were already written. However, during development, HBase 0.20 (for more information see Section 5.1) was starting to make an appearance and the performance figures looked very promising. So I decided to rewrite the EC2-scripts and started using Hadoop 0.20.0 and HBase 0.20.0-alpha instead.

If an experiment does not note the number of nodes used, it will usually be run on 1 master and 3 slaves. I used 'Large' EC2 instances for all servers (7.5GB RAM, 4EC2 Compute Units). If a memcached cluster is used in an experiment, it will be run on separate EC2 nodes of the 'Small' instance class, the cluster size will be 3 nodes.

I conducted my experiments on the MIRFLICKR-25000 image collection[18]. This collection is well documented and annotated, and is used in other image analysis project such as ImageCLEF 2009. This gives us the opportunity to compare our results to other experiments. It is however, seen from a web-scale perspective, very small. Therefore I will experiment with a very small cluster and expanding a bit, assuming that scaling the cluster will cope with scaling the image collection (the birth-right of any cluster solution).

5.1 HBase 0.20

The main goal for HBase 0.20 was simply to improve performance and reliability. A random row lookup in 0.19 could take well over 20 milliseconds, while the new 0.20 release reduces this to about 1-2 milliseconds. Mostly because of a complete overhaul of the data storage subsystem, which now also includes a result cache. Although the original design had a dedicated memcache cluster for caching, the HBase's internal result cache was deemed sufficient for this stage of the project. The direct result of the 0.20 release to my implementation was extremely positive. In early experiments with a color-only aspect index, a 3-node (+ 1 master node, also acting as Search Server) cluster, resulted in a 'term lookup' response time of less than 50 milliseconds when serving out of the HBase cache, and a response time of less than 120 milliseconds for any uncached query. HBase 0.19 did the same queries in over half a second.

Cluster Size	Webservers	Concurrency level	Request time[2]	Query time[3]	Requests/sec.
3	1	1	168	50	5.93
3	1	5	297	80	16.80
3	1	25	851	220	29.37
3	1	40	1171	380	34.15
		Tests performed from EC2			
5	2	1	62	30	16.10
5	2	5	134	60	37.80
5	2	25	393	220	63.48
5	2	40	608	270	65.79

Table 1: Performance of cluster on HBase 0.20

Using 2 webservers, the load of the Search Server program reaches 100% and thus should be about the limit for these specifications. The load on the region servers and the master however was minimal. So adding more Search Servers and webservers (using the proper ratio) should scale well. Also, there is a significant overhead on the webserver, sometimes even tripling the response time between the query and the client. This proves that a lot of improvement can be made by avoiding the expensive MVC stack of Zend Framework and using a lightweight client instead.

Hardware configuration of both Search Server and Webserver should be much bulkier in a production environment, like a 2x Quad Core with 16GB as Search Server, and 1x Quad-Core or 2x Dual-Core as webservers, thereby probably increasing Requests/second to well over 300 per second using the same number of servers.

6 Conclusion and Future Work

Because of the scale of this project, there are many opportunities left for future projects. To name a few, the indexing technique is still limited and unoptimized and incapable of dealing with a dynamic data set. This can be overcome by chaining several MapReduce jobs, e.g. an item count (for p_{db}-calculations), indexing job, collaborative filtering job, etc, instead of running just the indexing MapReduce jobs.

[2] Average time for full (remote) HTTP request in milliseconds, 20 seconds long

[3] Average query time at Search Server, over the same 3 runs

Also, the indexes themselves need work; there is no hierarchy in the index yet, making a serious impact on performance because the posting lists are far too large to efficiently query the engine. Another problem in that area would be to adjust ARL to handle a multi-layered index.

Other obvious improvements are those mentioned in the report itself: a crawler, relevance feedback system, funding for a permanent cluster location, etc. I hope to implement these features in other projects in the course for my Masters degree.

On a personal note, the project also taught me an interesting approach to databases and information retrieval and introduced me to the world of 'schema-free, distributed, nosql'-databases. Finally, HBase inspired me to look at (and hopefully work with) databases like CouchDB and Katta for future projects.

References

[1] Michael Lew. Content-based multimedia information retrieval: State of the art and challenges. *ACM Transactions on Multimedia Computing, Communications, and Applications*, pages 1–19, 2006.

[2] M.J. Huiskes. Image searching and browsing by active aspect-based relevance learning. *CIVR 2006*, pages 211–220, 2006.

[3] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 1998.

[4] MySQL. `http://www.mysql.com`.

[5] Database Sharding at Netlog, with MySQL and PHP. `http://www.jurriaanpersyn.com/archives/2009/02/12/database-sharding-at-netlog-with-mysql-and-php/`.

[6] Database sharding blog. `http://www.codefutures.com/weblog/database-sharding`.

[7] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *OSDI*, 2004.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *OSDI*, 2006.

[9] David DeWitt and Michael Stonebraker. Mapreduce: A major step backwards. http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/.

[10] Hadoop at Yahoo! `http://developer.yahoo.net/blogs/hadoop`.

[11] Hadoop at Rackspace. `http://blog.racklabs.com/?p=66`.

[12] Powerset, leveraging open source hadoop, powers microsoft's bing. `http://ostatic.com/blog/powerset-leveraging-open-source-hadoop-powers-microsofts-bing`.

[13] Apache Zookeeper. `http://hadoop.apache.org/zookeeper`.

[14] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. *OSDI'06*, 2006.

[15] Apache HBase. `http://hadoop.apache.org/hbase`.

[16] Apache Mahout - Taste. http://lucene.apache.org/mahout/taste.html, 2008.

[17] Apache Pig. http://hadoop.apache.org/pig.

[18] M.J. Huiskes and M.S. Lew. The MIR Flickr retrieval evaluation. *ACM International Conference on Multimedia Information Retrieval*, 2008.

[19] Flickr. http://www.flickr.com.

[20] Dojo toolkit. http://www.dojotoolkit.org.

[21] Dojo drag and drop, part 1. http://www.sitepen.com/blog/2008/06/10/dojo-drag-and-drop-1.

Appendices

A MapReduce pseudo-code

```
map(key, line,
  OutputCollector<Text, IntWritable> output) {

  String line = new SplitLineToWords(line);
  while (line.hasMoreWords()) {
    output.collect(line.nextWord, 1);
  }
}

reduce(key, Iterator<IntWritable> values,
  OutputCollector<Text, IntWritable> output) {
  // sum all values for this key
  long sum = 0;
  while (values.hasNext()) {
    sum += values.next().get();
  }

  // output sum
  output.collect(key, new IntWritable(sum));
}
```