2012

Large Scale Image Retrieval From Books

Mao Zhao
University of Massachusetts Amherst

LARGE SCALE IMAGE RETRIEVAL FROM BOOKS

A Thesis Presented

by

MAO ZHAO

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2012

Electrical and Computer Engineering

LARGE SCALE IMAGE RETRIEVAL FROM BOOKS

A Thesis Presented

by

MAO ZHAO

Approved as to style and content by:

_____
James Allan, Chair

_____
Lixin Gao, Chair

_____
Aura Ganz, Member

_____
C.V. Hollot, Department Chair
Electrical and Computer Engineering

ABSTRACT

LARGE SCALE IMAGE RETRIEVAL FROM BOOKS

SEPTEMBER 2012

MAO ZHAO

B.E. BEIJING INSTITUTE OF TECHNOLOGY

M.S.E.C.E. UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor James Allan, Professor Lixin Gao

Search engines play a very important role in daily life. As multimedia product becomes more and more popular, people have developed search engines for images and videos. In the first part of this thesis, I propose a prototype of a book image search engine. I discuss tag representation for the book images, as well as the way to apply the probabilistic model to generate image tags. Then I propose the random walk refinement method using tag similarity graph. The image search system is built on the Galago search engine developed in UMASS CIIR lab.

Consider the large amount of data the search engines need to process, I bring in cloud environment for the large-scale distributed computing in the second part of this thesis. I discuss two models, one is the MapReduce model, which is currently one of the most popular technologies in the IT industry, and the other one is the Maiter model. The asynchronous accumulative update mechanism of Maiter model is a great fit for the random walk refinement process, which takes up 84% of the entire run time, and it accelerates the refinement process by 46 times.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Overview of image retrieval and books project

Image search engines play an important role in providing people desired pictures and straightforward image information. Existing image search schemes can be roughly divided into three types: text-based search, similar image search, and sketch-based search [1]. Most of the existing image search engines are web image search systems, and they can be classified into different categories in terms of how images are represented. This can be done since web images usually come along with HTML source code including textural descriptions. Many web image search systems are text-based and the representation of the images includes filenames, captions, surrounding text, etc. These systems measure the similarity between image and user query by estimating the probability that the corresponding textual information of the image is relevant to the query [2]. Some applications use tags as descriptive words for images [3]. Popular commercial image search engines today like Google Image Search and Microsoft Live Image Search rely mostly on surrounding text features and click logs. Considering that the textual information associated with web images might be noisy and incomplete, some systems try to improve web image search performance by leveraging visual features [4]. Thus, besides of the text retrieval approach for image search, a Content-Based Image Retrieval (CBIR) approach from the computer vision aspect is proposed. CBIR relies on machine perception, a way of characterizing images based on their visual content. CBIR

systems analyze the actual contents of the image like colors, shapes, textures, or any other information that can be derived from the image itself [5].

The Million Book project is a book search engine being built in collaboration with the Internet Archive Digital Library by University of Massachusetts Center for Intelligent Information Retrieval (CIIR) research group. The basic goal of this book search engine is to gather the most relevant book pages for a certain search query, i.e. search for “King Lear” or “Hamlet”, should result in pages of Shakespeare’s related work should be pulled out and ranked by relevance. The Book project is being built on Galago, an open source search engine developed by CIIR research group [6]. About 5 million book pages have been indexed, and basic search can be performed. People in this project are working on advanced features to make this book search engine more fun, which will be briefly talked about in several interesting subprojects here. One subproject, which I am working on, is the large-scale book image search. The goal of the large-scale book image search is to find out images that are very relevant to the query from a very large book collection. The image retrieval and cloud computing techniques are the main concern of this thesis. I discuss book image retrieval in the aspects of “book image representation”, “image ranking”, and “image ranking refinement”. Then I discuss the cloud computing for image retrieval in the aspects of “scale issue”, “accumulative updates”, and “Maiter framework deployment”. In the end, I will show the evaluation and experiment results. Another interesting subproject is location entity retrieval, which enables users to search for stories that happened in a given location. For example, when a search for the location “Caribbean” occurs, the page that talks about the story of

Caribbean pirate should be displayed to users. Other subprojects including evidence finding, name entity disambiguation and duplicate finding all add fancy feature to the book search engine.

1.2 Overview of scale issue and cloud computing

Search engines are designed to retrieve information from large amount of data, so the scale issue is one of the most significant problems to deal with. Commercial search engines like Google and Microsoft Bing collect websites all over the world periodically, and therefore, data centers are necessary for the heavy computation tasks. Our book search engine is not crawling data all over the world, but 5 million books is huge for research purpose. In my prototype, I have a dataset of 50 books, which is 100 Megabytes, and we can roughly estimate that 5 million books would be 10 Terabytes. Experiment shows that the most time consuming part in the indexing process, the random walk refinement, takes more than 3 hours for 100 Megabytes data. Therefore, if we cannot figure out how to reduce the execution time significantly, more than 36 years are needed to index all the 5 million books. The most straightforward thought to solve this scale issue is distributing the job to more machines and execute in parallel. This thought leads us to introduce the cloud computing techniques into our book search engine.

Cloud computing, one of the most popular technologies nowadays, makes it possible for companies to provide more attractive software and services. Cloud computing refers to both the software applications services and the system software that provides these services. We call the former “Software as a Service” (SaaS), and

the latter “Cloud” [7, 8]. Amazon EC2 is one of the commercial cloud computing platforms for which users pay to rent EC2’s hardware virtual machines. Users take advantage of the batch processing ability to process terabytes of data in a relatively short time as long as the job has enough parallelism [7]. Companies who can provide the “Cloud” not only need to operate a data center but also have to develop software infrastructure. Some companies use MapReduce, a programming model designed for large-scale dataset, as the software infrastructure [7, 9]. For example, Amazon Web Service platform team use Hadoop framework for their development. To sum up, we can deploy all the time consuming part in the image search engine on the cloud through some software infrastructure, such as MapReduce, to accelerate execution. Figure 1 shows basic concept.

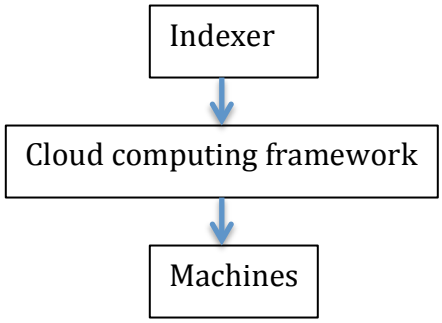


Figure 1 build indexer on cloud

1.3 My approach

For image retrieval, the first question people may ask is how to represent an image. Since the collection is a set of image pages within books, and most of the books are old, it is not a good idea to use the visual representation. Consider the large amount of information books contain, I am going to extract descriptive words or phrases to represent images. I call these descriptive words or phrases “tags”.

Therefore, a list of tags could be used as the representation of the image. The quality of tags is the key factor for image retrieval. The second question is how I can extract descriptive tags that are relevant to the image. I use the probabilistic approach for tag generation, which I will discuss in chapter 3, and then I will introduce the similarity clustering method for tag list optimization. By performing a random walk on a tag graph according to similarity cluster, the importance of the unique tags will be weakened, while the importance of "similar" tags will be enhanced. Similarity clustering and random walk refinement is discussed in chapter 4.

For cloud computing, the first question that comes to my mind is why I use this technique. Chapter 5 does the efficiency analysis and execution time estimation based on experiments with relatively small dataset, and convinces readers that we do need the cloud computing technology. According to chapter 5, we notice that the random walk process takes up to 84.5% of the entire execution time, and when the number of books gets very large, the scale problem becomes critical. Therefore, we are clear about the goal: make random walk execute faster with the help of cloud computing model. I try to solve the scale problem with two different models, the MapReduce model and the Maiter model. The MapReduce model is kind of similar to functional programming and it relies on Map and Reduce functions to process key value pairs. There are implementations of MapReduce in different programming languages, and the most commonly used framework is Apache Hadoop [10], which I have been using in my experiment. Maiter model is totally different with MapReduce model from the underlying mathematical basis. Maiter can do an accumulative update without waiting for the previous iteration to finish, and we call it

"asynchronous accumulative update". Compared to MapReduce, which cannot step into the next iteration before the previous iteration is done, the asynchronous update feature will save a lot of time and will speed up the processing. I discuss the principle of these two models in chapter 6, and do a comparison to choose one that fits my search engine better. Chapter 7 talks about the implementation and deployment of the Maiter model. Experiments are also demonstrated in this chapter and you will see what a big favor Maiter gives me.

The last question is how effective is my image search engine. Evaluation is very important to judge the search quality. I use the mean average precision for measurement. Evaluation is discussed in chapter 8.

1.4 Contributions

- I build the book image parser for book page filtering, pre-processing and image indexer based on the Galago.
- I build the TupleFlow stages for tag list generation, and use probabilistic model on the tag list to generate initial tag relevance score.
- I build the similarity graph for the whole collection and tag-similarity graph for every image page, and transplant random walk model to refine the tag list.
- I re-build the random walk refinement based on both the MapReduce model and Maiter model, and prove the correctness of building random walk with accumulative update model.
- I do experiments on the comparison between the performance of re-built random walk refinement on the MapReduce model and Maiter model.

- I deploy the random walk refinement on the Maiter framework and do experiments on the time consumption.
- I index 50 books, 4736 image pages, and do an evaluation on 135 pages. The mean average precision increases from 28% to 30.4%, and the execution time decreases by 46 times.

1.5 Thesis organization

This thesis begins with a discussion of the high level idea of how to build an image search engine for books in chapter 2. The key concept “image tagging” is discussed in detail. Fundamental information retrieval concepts and techniques, such as inverted index construction, term weighting, document ranking and query language, are also discussed. Chapter 2 and chapter 3 talk about how to extract descriptive words as “tags”, using a probabilistic model and bonus feature for tags. Chapter 4 is the most important one in this thesis. Similarity graph and random walk refinement are discussed, and the effectiveness of random walk refinement makes a huge influence on search quality. Chapter 5 analyzes the efficiency, defines the scale issue and introduces the cloud computing. Chapter 6 focuses on MapReduce model and Maiter model, and discusses functional programming and accumulative update accordingly. Experiment results occur with the comparison of these two models, which are given in this chapter as well. Chapter 7 mainly explains how to take advantage of Maiter to run the random walk job. Chapter 7 gives very convincing experimental results (speed up 46 times) that could prove Maiter cloud computing model is a very good fit for image search engine. Chapter 8, the

evaluation part, tells reader how effective the system is and whether the random walk process improves the performance or not.

CHAPTER 2

BOOK IMAGE SEARCH

2.1 Fundamental text retrieval techniques

2.1.1 Indexing

2.1.1.1 Terms, stopping words and stemming

To begin with, let me introduce the concept of term and token. A term is a unique word in the dictionary. For example, the sentence “computer in computer science department” contains four terms, because computer appears twice, they are the same term. Now let me chop this sentence into pieces, and we get an output like this: “computer”, “in”, “computer”, “science”, and “department”. We call each piece a “token”, and this sentence has five tokens. Terms and tokens are the basic evidence for text retrieval. Sometimes, extremely common words are excluded from the vocabulary entirely. These words are called stop words. We use an existing stop list to get rid of stop words, and it significantly reduces the number of postings that the system has to store. Also notice that “dog” and “dogs” are actually the same word, but search engine does not know. A technique is needed to convert them into the same thing. An algorithm called “stemmer” can do this. Such an algorithm assigns natural language words to stem classes, which are groups of words that are presumed to represent the same concept. A stemmer can be either aggressive or conservative. The former one tends to mistakenly group words that should not be grouped, and the latter one may fail to group related words [6]. Suffix-s stemmer provides the simplest kind of English stemming. This stemmer only conflates plural

words with their singular versions. This type of stemmer is not aggressive and works well in practice. In the Galago, we use Porter2 stemmer, which determines classes of similar words on the basis of suffix patterns [6]. In books image search project, all tags are stemmered.

2.1.1.2 Inverted index

We have discussed terms in 2.1.1.1, but terms cannot be the minimum retrieval unit, because we do not want a list of terms to be returned as the search result. We define the minimum retrieval unit as a "document". You can say that each search result is a ranked list of documents even though it is not precise. A document cannot be too long. Specifically for the book project, it is a bad idea to index the whole book as just one document. We use one book image page as one document. To rank the documents for a specific query, we need to know how important this query is to each document, and that leads us to define the measurements. The term frequency is the most straightforward measurement for the query importance. We denote the term frequency in the document as f_d, and the term frequency in the collection as f_c. The term position is also an important feature for text retrieval. It is very helpful if we are searching for phrases. For example, we can search for the phrase "Tom Cruise". One document has the name "Tom Hanks" that appears 20 times. Another document has the name "Tom Cruise" that only appears 3 times. The query term "Tom" in the first document occurs more often than in the second document, thus by counting the term frequency, the search engine may say that the first document is more possible to relate to "Tom Cruise" than the second one, but

the truth is on the contrary. Let's see what happened if we bring in position to the search engine. Now we tell the search engine: if each term in the query appears in the document sequentially, mark that document as "very important". The position feature is used to locate terms by the search engine. The search engine can tell whether two terms are neighbors by checking their positions. Let's go back to the previous example. The second document will be marked as "very important" this time according to the position rule. In practice, we not only use neighbor terms to emphasize document importance, but also use position range to add value for the documents. For example, let's say some query terms appear within 2 term-distance (next to each other), and let's say this will be attributed with the bonus value of 10, while some other query terms appear within 10 term-distance may be assigned a bonus value of 5. The potential usage of term position (denoted as "tag position" in image search project) in the book image search project is different from normal use. It can be used to extract phrases, and it can be used to compute the distance between tags and images. Know the distance of tags and images is important if one books page contain multiple images, because we need to tell which block of text describe which image.

Till now, we have held the basic elements to build an index: terms, term frequency, documents and term position. Term position is not necessary, but it is a good feature to improve search quality. An inverted index should have a list of terms first. Note that when we say term, that means it is unique. The list of terms cannot have any duplicate words, and it should be like a dictionary. For each term, we need to know which documents it appears in. This leads us to build a posting list for each

tag, which stores document id and term frequency f_d. Figure 2 shows the basic structure of the inverted index.

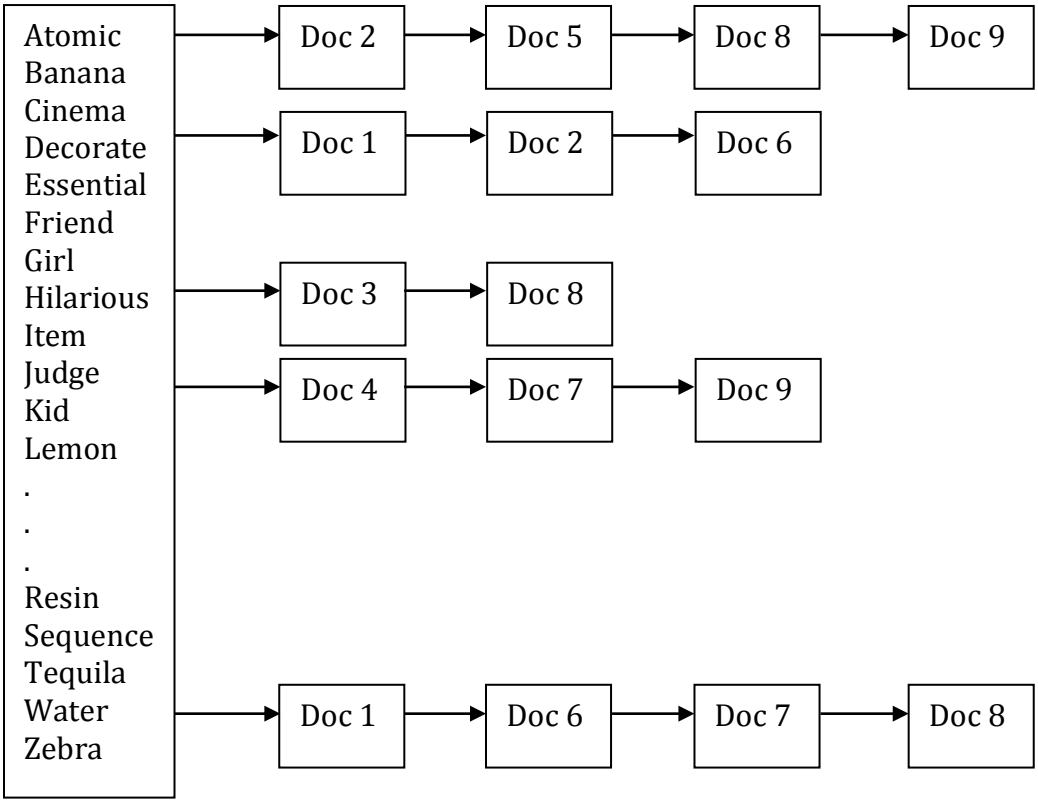


Figure 2 posting list example

2.1.2 Term weighting and document ranking

Sometimes certain terms have limited power in discriminating relevance. For example, a collection of documents on the entertainment industry is likely to have the term "entertainment" in almost every document. Therefore we need to lower the term's weight with a high collection frequency. Collection frequency is defined to be the total number of occurrences of the term in the collection. It is more common to define document frequency to be the number of documents in the collection that contains certain terms. Then the inverse document frequency is derived to scale the terms' weight [11].

$$\mathrm{idf}_t = \log \frac{N}{\mathrm{df}_t} \quad (2.1)$$

Combine term frequency and document frequency, we get tf-idf weighting as follows [11].

$$\mathrm{tf} - \mathrm{idf}_{t,d} = \mathrm{tf}_{t,d} \times \mathrm{idf}_t \quad (2.2)$$

Highest tf-idf value is reached when the term appears many times within a very small number of documents, and vice versa for the lowest tf-idf value. Query's weighting is the summation of all terms' tf-idf score in that query [11].

$$\mathrm{score}(q, d) = \sum_{t \in q} \mathrm{tf} - \mathrm{idf}_{t,d} \quad (2.3)$$

2.1.3 Query

In section 2.1.1.2, we say that term position is a good feature to improve search quality. I would like to discuss a little bit more on this by introducing indri query language developed by Strohman [6]. For example, query "Lady Gaga" could be translated into the following indri language.

```
#weight ( 0.8 #combine ( #wsum ( 1.0 Lady.(mainbody)
                                 3.0 Lady.(title)
                                 2.0 Lady.(heading))
                         #wsum ( 1.0 Gaga.(mainbody)
                                 3.0 Gaga.(title)
                                 2.0 Gaga.(heading)))
          0.2 #combine ( #wsum ( 1.0 #1( Lady Gaga).(mainbody)
                                 3.0 #1 (Lady Gaga).(title)
                                 1.0 #1 (Lady Gaga).(mainbody))))
```

Figure 3 query language example

This piece of query language means: weighting for query "Lady Gaga" has two parts. For the first part, we weight for each term "Lady" and "Gaga" separately.

Terms that occur in the title are assigned more weight than terms that occur in the heading, and terms occurs in the heading are assigned more weight than in the mainbody. For the second part, if “Lady Gaga” occurs as a phrase, we assign bonus for this query. Similar to the first part, “Lady Gaga” in the title gets the highest weight.

2.2 Prototype

Images on the web have more available resources than images in books for a search engine. Anchor text and link analysis could be applied on the web image search, however, books do not have these advantages to get relevant information, and all the information needed for image retrieval is obtained from the book pages. Therefore, we try to extract some words or phrases that could describe the content of the image as search evidence. One obvious thing is that the captain, title and words in the image would be great evidence for image retrieval. However, not all images have a title or captain, these features could only be considered as very good bonus for the search. Our principal evidence still sticks to the descriptive words in the surrounding text. Notice that if people’s name or place name appears in some image pages, it is very likely that the image is just that person or place, and therefore these names are very good evidence. We call people’s name or place name “entity”. Now that we know the search evidence for images, we can build a prototype for image search engine.

From the highest level of view, the prototype consists of five components, which are the search interface, indexer, cloud computing framework, storage system

and the evaluation system. Figure 4 illustrates the flow chart of these components. The book database stores millions of OCR'd books, but it is not part of the image search engine prototype.

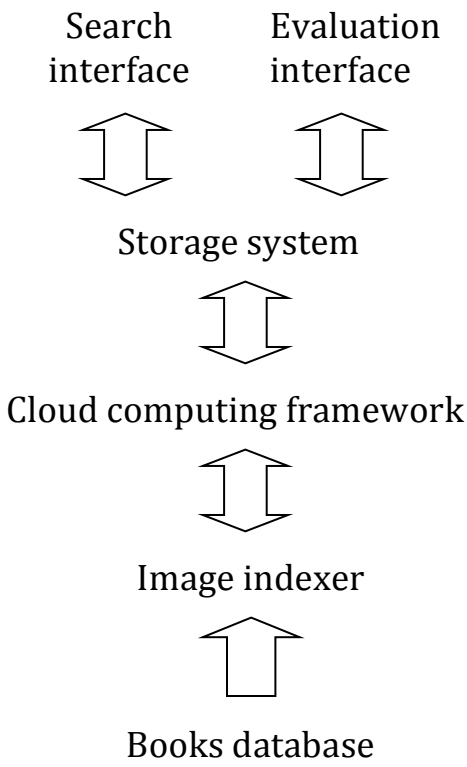


Figure 4 basic system modules

Book pages will be processed offline. The image indexer, which is deployed on cloud computing framework, fetches book files from book databases, processes book pages, and write an index map as well as tag list files into the storage system which is a structure designed to speed up query retrieval. The search interface takes in query, retrieve information from the storage system, and give results back to the interface. The evaluation system retrieve image pages from the index storage system first, then collect evaluation information provided by users, store the result back into the storage system and compute the mean average precision. The search

interface consists of a web interface at the front-end and a query evaluator (query parser) at the back-end. The evaluation interface consists of a web interface at the front-end and a precision evaluator at the back-end. The storage system consists of index maps, object files, similarity graphs and tag ranking lists. The image indexer is built on Galago and deployed on Maiter cloud computing framework. The following chapters talk about detailed models, such as TupleFlow in Galago and accumulative update in Maiter.

2.3 Image tagging

2.3.1 Image representation

Images could be represented in different ways. From the image processing point of view, it could be represented by a matrix of pixels in the time domain and a sequence of digital values in the frequency domain. If any visual approach is used, pixel matrix representation should be useful. However, as I discussed in section 1.1 and section 2.2, content-based image retrieval (image retrieval by visual feature) does not work for images in book pages, so tag representation makes more sense.

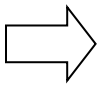
2.3.2 Tag ranking

2.3.2.1 Rank by tag relevance

Image tags are a bunch of words that describe the image. Some tags are very relevant to the image while some are not that descriptive. Thus, we introduce “tag relevance” to represent how descriptive the tag is. Tags with high “tag relevance” lead to high quality search, and vice versa. Therefore, low relevance tags should not

be considered important in search. This could be done easily by ranking the tag list according to "tag relevance" in descending order. Figure 5 is an example of expected tag ranking. More relevant tags like "aircraft", "fly", "sky" has their position in the ranking list moved up.

Wheel
Boeing
Basketball
Fly
Pilot
Aircraft
Sky

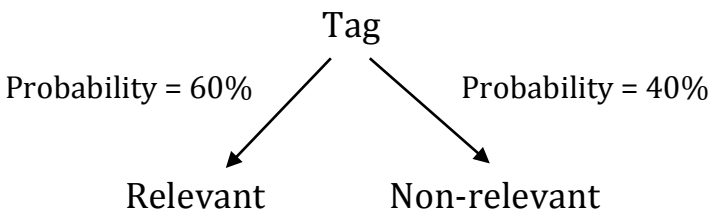


Aircraft
Fly
Boeing
Basketball
Pilot
Sky
Wheel

After relevance ranking

Figure 5 tag list example

Now we know "tag relevance" is important, but you may ask what the mathematical representation of "tag relevance" is and how we can compute the "tag relevance". Assume that each tag has only two statuses: "relevant" and "non-relevant", each tag has the probability to be in either one of the statuses. We assign the probability that the tag falls in to "relevance" status to be its "tag relevance" score. Figure 6 illustrates this. In Figure 6, the "tag relevance" is 0.6.



"Tag relevance" = 0.6

Figure 6 tag relevance

We use the probability representation for "tag relevance", and we discuss how the probabilistic scores are generated for each tag in chapter 3.

2.3.2.2 Rank by tag similarity

Another import concept is “tag similarity”, which evaluates how similar two tags are. For example, let’s assume that there are five tags, “aircraft”, “pilot”, “fly”, “sky” and “basketball”. Obviously, “aircraft”, “fly”, “sky” and “pilot” are more related and therefore should be clustered into the same group. Figure 7 illustrates the similarity clustering.

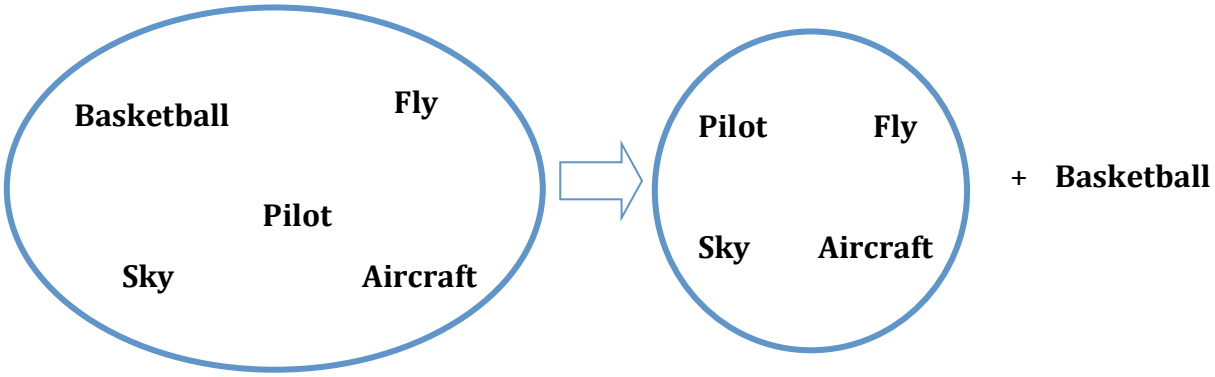


Figure 7 similarity clustering

Unique tags tend to be less descriptive, so it is a good idea to weaken the “tag relevance” for such tags. Let’s continue the example in Figure 5. Figure 8 illustrates the expected goal of ranking refinement by similarity clustering.

Wheel	Aircraft	Aircraft
Boeing	Fly	Fly
Basketball	Boeing	**Sky**
Fly	**Basketball**	Boeing
Pilot	Pilot	Pilot
Aircraft	**Sky**	Wheel
Sky	Wheel	**Basketball**
		After similarity clustering refinement

Figure 8 refinement by similarity clustering

2.3.2.3 Feature of image tags

Try to recall the “anchor text” in the web search. The “Anchor text” would be considered really important by the search engine since it could be a very good summary of a web page and it is usually very informative. In book image search, there are such “anchor text” as well. Captains or the “small descriptive word” under the image are usually very descriptive and informative, and we give them a name “anchor tags” Figure 9 gives an example of these important tags in book image pages.

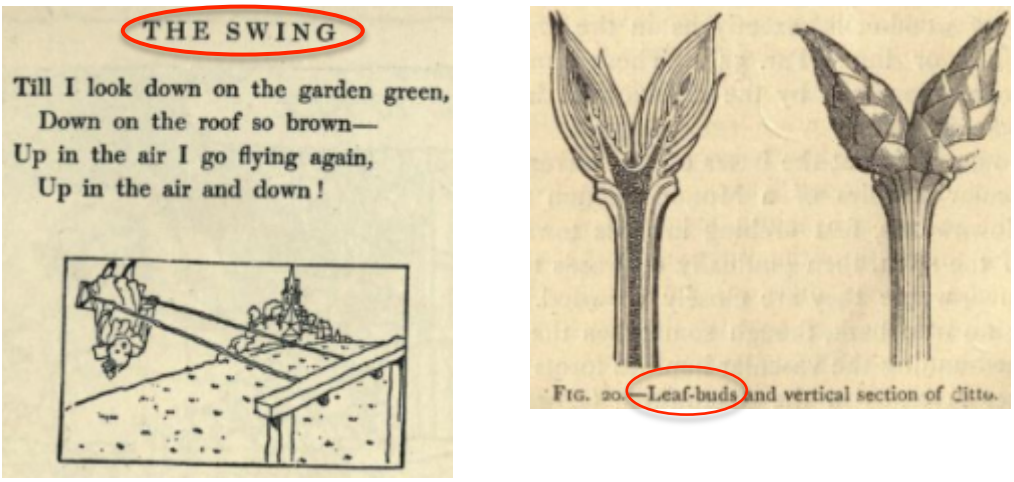


Figure 9 title and caption

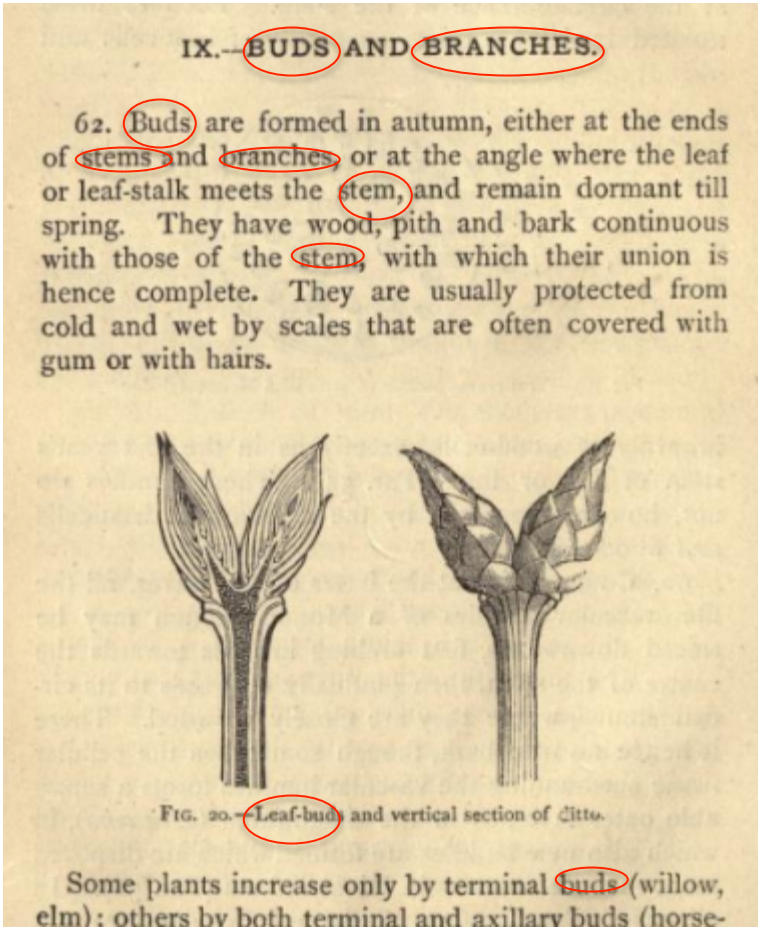
As we can see, “swing” and “leaf buds” in Figure 9 are exactly what the image shows. Thus, in the retrieval model, we should consider the “anchor tags” to be very important evidence.

CHAPTER 3

TAG GENERATION

3.1 Probabilistic model for image tagging

As discussed in section 2.3, image tags are descriptive terms, and a list of tags represents the image. In this chapter, we discuss in detail how the tag lists are generated and how to refine them to produce more descriptive tags. According to the evidence analysis in section 2.2, tags are extracted from page text. Let's take a look at the surrounding text in Figure 10 first.



IX.—BUDS AND BRANCHES.

62. Buds are formed in autumn, either at the ends of stems and branches, or at the angle where the leaf or leaf-stalk meets the stem, and remain dormant till spring. They have wood, pith and bark continuous with those of the stem, with which their union is hence complete. They are usually protected from cold and wet by scales that are often covered with gum or with hairs.

FIG. 20.—Leaf-buds and vertical section of ditto.

Some plants increase only by terminal buds (willow, elm); others by both terminal and axillary buds (horse-

Figure 10 image tagging

Terms in red circles are selected as tags. People's opinion may vary, and in my point of view, "bud", "branch" and "stem" are the best descriptive terms for the image in Figure 10. Given an image page D, we assume that tags are generated by the text on that page with some probability. This probability could be roughly evaluated by the following equation [12].

$$P(t|D) = \frac{c(t; D)}{|D|} \quad (3.1)$$

c(t;D) is the count of tag t in image page D, and |D| is the length of image page D. Under the bag of words assumption, we assume that there is no need to model tag dependence. Since tags are independent, the probability that tag t1, t2, t3 … tn are all generated by image page D is the product of the probability of each tag.

$$P(t1, t2 \dots tn|D) = \prod_{1<i<n} P(ti|D) \quad (3.2)$$

Tags generated with high probability are considered to be the key tags. They are more likely to relate to the topic. Generally speaking, images are always the topic of the page it sits on. For example, looking at the book page in Figure 10, we see the whole page talks about buds and branches.

From the above analysis, we derive the basic probabilistic model for tag generation: rank tag list by probability of generation. Intuitively, if the tag list has been ranked by probability, it is also ranked by relevance. Equation (3.2) could be extended to estimate the probability of a given query generated by a specific document.

$$P(Q|D) = \prod_{t \in Q} P(t|D) \quad (3.3)$$

In equation (3.3), Q is the given query, and t is the term in query Q. P(Q|D) is estimated to be the probability that query Q be generated by page D. However, this basic probabilistic model has two problems. First, if term t appears in query Q but not in image page D, then P(Q|D) = 0. Put it another way, the image page that does not contain all of the query terms cannot ever be relevant to the query, or, the image page that contains some of the query terms is not any more likely to be relevant than a document that contains no query term [6]. Second, all terms in the query are treated identically. Fortunately, Smoothing can solve both of the problems. We assume that image page D is just a sample from the whole image page collection, and the whole collection could be treated as large natural language text. The best sample of natural language text is the entire text of our document collection. We define the collection probability as follows.

$$P(t|C) = \frac{c(t; C)}{|C|} \quad (3.4)$$

c(t;C) is the count of tag t in the whole collection C. we add this additional model into the basic probabilistic model to smooth P(t|D). A straightforward way to combine these two models is doing a linear combination with a parameter λ, where $0 < \lambda < 1$. We redefine the probability of tag t in the image page D smoothed by collection probability as follows [6, 12].

$$P(t|D) = (1 - \lambda)\frac{c(t; D)}{|D|} + \lambda\frac{c(t; C)}{|C|} \quad (3.5)$$

Another approach for smoothing is to assume model of natural language text generates model for D. Since the model we are using for text is multinomial, the natural generating distribution is the Dirichlet distribution. Estimate the likelihood

of P(t|D) given that D is generated by a Dirichlet distribution with a parameter vector set to the collection distribution, we get another expression for the probability of tag t in image page D as follows [6].

$$P(t|D) = \frac{c(t; D) + \mu c(t; C)/|C|}{|D| + \mu} \quad (3.6)$$

In equation (3.6), D is modeled by |D| tags in the image page plus μ addition tags drawn random from the whole collection [6]. In equation (3.3), the right part could be estimated by either (3.5) or (3.6). To estimate a given query, we take a log at both side of equation (3.3), because the probability could be very small. The product in (3.3) becomes summation of logs.

$$\log P(t|D) = \sum_{t \epsilon Q} \log P(t|D) \quad (3.7)$$

Equation (3.5) and (3.6) could be used as tag generation model, and we choose equation (3.5) in book image search project.

3.2 Tag relevance refinement

3.2.1 Relevance model

In the discussion above, we have defined tag relevance to be the probability of tag t generated by image page D smoothed by collection probability. Now I would like to talk about the possible way to improve this model. In section 2.1.2, we have introduced concept of inverse document frequency and vector space model for document ranking, and the idea of adding vector space model feature to the probabilistic model come to my mind. Notice that text on image pages usually has a focus on a specific topic, which is about the image on that page, so we can say that

the most relevant tags are unlikely to occur on every page in the collection. Descriptive tags should appear in a small portion of the pages that focus on relevant topics. Therefore, for tags that occur in a lot of image pages, tag relevance should be weakened. As discussed in section 2.1.2, inverse document frequency is a good match for this purpose. Recall that $\mathrm{idf_t} = \log\frac{N}{\mathrm{df_t}}$, we propose a model that combine probabilistic model and inverse document frequency feature, and we give it a name "Relevance Model".

$$R(t; D) = \mathrm{idf_t} \times P(t|D) \quad (3.8)$$

Combine equation (2.1), (3.5), (3.8), we get the relevance model equation.

$$R(t; D) = \log\frac{N}{\mathrm{df_t}} \times \left[(1-\lambda)\frac{c(t; D)}{|D|} + \lambda\frac{c(t; C)}{|C|}\right] \quad (3.9)$$

R(t;D) is the tag relevance of tag t in image page D. N is the total number of pages in the collection, $\mathrm{df_t}$ is the document frequency of tag t. λ is the probabilistic parameter, |D| is the length of image page D, |C| is the collection length, c(t;D) is the count of tag t in image page D, and c(t;C) is the count of tag t in the whole collection. "Relevance Model" has not been implemented yet, and current implementation in book search project is still probabilistic model.

3.2.2 Bonus feature

Many different kinds of features are possible. We introduce three approaches that are good for book image project: noun phrases, word proximity and weight of different parts of a document differently. First, noun phrases like a person's name or place's name are usually important evidence for retrieval and should be assigned

more weight. In the image search project, entities are all noun phrases. For example, “Koala bear” is a noun phrase. In book image search, each term has an attribute that tells you this term is a noun or adjective. We can construct a sentence tree according to this attribute. For example, if we have a sentence “A very cute Koala bear climbs slowly onto a big tree”, it could be represented in a tree structure in Figure 11.

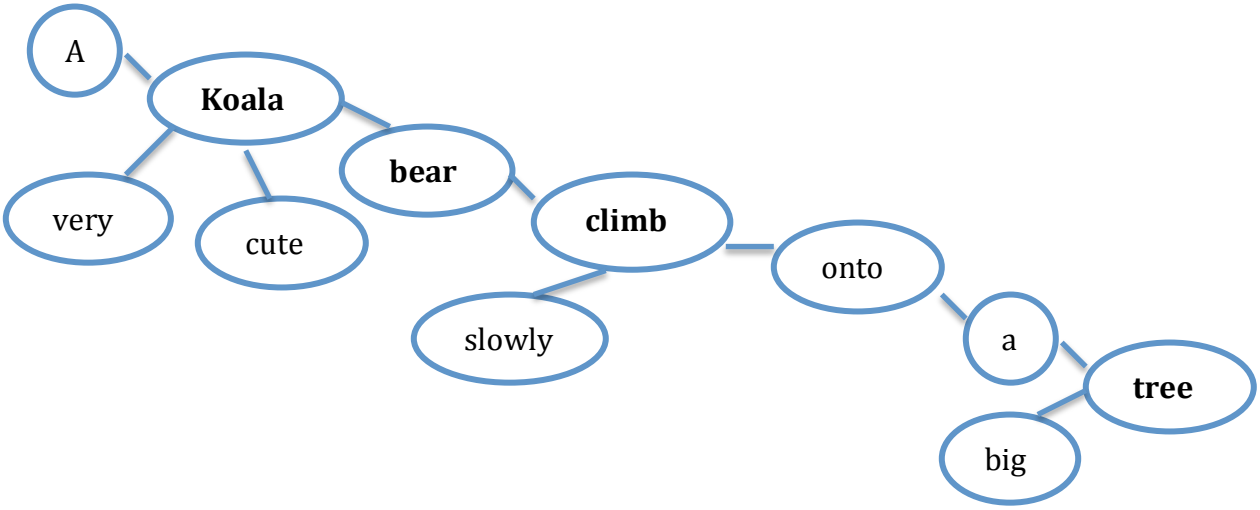


Figure 11 phrase tree

In Figure 11, “Koala”, “bear” and “tree” are nouns, “very” and “cute” are adjectives that describe “Koala”, “big” is an adjective that describe “tree”, “slowly” is an adverb that describe the verb “climes”, and “a” as well as “onto” are stop words. Eliminating the stop words, adjectives and adverbs, we get the stem of the sentence. Figure 12 illustrates the result. If we get rid of verbs, then only “Koala bear tree” left.

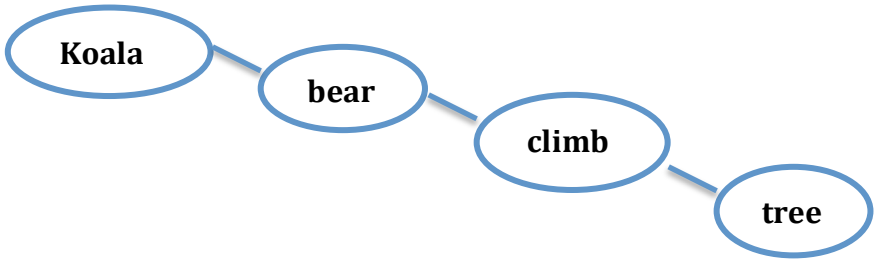


Figure 12 sentence stem

How can we make the phrase “Koala bear” a single unit, just like one term? This question leads us to the second feature, word proximity.

Word proximity is applied on query, not tags. Although it is not relevant to tag generation, I would like to give a brief explanation about how it works. Follow the “Koala bear” example. It is obvious that the phrase “Koala bear” is a much better indicator than the separated “Koala” and “bear”. Metzler and Croft’s Markov Random Field model [13] mentioned single term features, exact phrase features and unordered window features. Exact phrase features said if terms in query occur one next to another in order, document D would be assigned bonus weight. Unordered window feature said that as long as the query terms occur within some window distance, document D would be assigned bonus weight. Figure 3 in section 2.1.3 is an example of the word proximity feature.

CHAPTER 4

RANDOM WALK RANKING REFINEMENT

4.1 Similarity clustering

In section 2.3, we have discussed the reason to introduce the concept of tag similarity, and how it improves the tag ranking. In this chapter, I will talk about the similarity graph construction and ranking list refinement with the similarity graph. First of all, let's recall the probabilistic model assumption: all image page D is generated from the Dirichlet distribution (nature language text). If the probability that page D contains both tag T1 and tag T2 is high, we say that T1 and T2 have high similarity. Now we can define tag similarity in math as follows.

$$s(t1, t2) = P(D|t1, t2 \in D) \quad (4.1)$$

Equation (4.1) means that the similarity between tag t1 and t2 equals to the probability that image page D been generated by Dirichlet distribution given that t1 and t2 are in D. The probability could be computed as follows.

$$P(D|t1, t2 \in D) = \frac{c(D_{t1,t2})}{N_D} \quad (4.2)$$

$c(D_{t1,t2})$ is the count of pages that contain both tag t1 and t2, and N_D is the total number of pages. This probabilistic representation for tag similarity is not perfect. Consider this example: we have five pages in Figure 13, we compute the similarity of "Dog" and "Puppy", and the similarity of "Animal" and "Cat" using equation (4.1) and (4.2). s(dog, puppy) = 1/5 = 0.2, and s(animal, cat) = 2/5 = 0.4. However, the "dog" and "puppy" are obviously more similar, and should get higher similarity score than the words "animal" and "cat".

Figure 13 similarity clustering example

This happens because some tags are popular, or we say that they are general tags, such as "animal". The general tags may occur in more pages than the specific tags, such as "dog". Therefore, the probability that general tag shows up is always high. We scale down the effect of the general tag by taking advantage of the high probability it occurs alone. Now we define the rules to give tags high similarity.

Tag t1 and t2 are assumed to be very similar if they satisfy the following rule

(1) Probability that tag t1 and t2 appear in the same image page is high.

(2) Probability that tag 1 and tag 2 appear alone in different image pages is low.

We re-define the similarity as follows [3].

$$d(t1,t2) = \frac{\max\left[\log\left(f(t1)\right),\log\left(f(t2)\right)\right] - \log\left(f(t1,t2)\right)}{\log(N_D) - \min\left[log\left(f(t1)\right),log\left(f(t2)\right)\right]} \quad (4.3)$$

$$s(t1,t2) = \exp\left(-d(t1,t2)\right) \quad (4.4)$$

Apply equations (4.3) and (4.4) to the above example in Figure 13, we have s(dog, puppy) = exp(-0) = 1, and s(animal, cat) = exp(-0.67) < 1 = s(dog, puppy). This time, it makes more sense. Now given an image page, we can easily cluster tags according to the similarity. For system use, we build a tag similarity graph. As showed in Figure 14, it looks like a posting list.

The next question is how to take advantage of the tag similarity to improve the tag relevance ranking. Here, we use the random walk approach to recursively

refine the tag relevance score, and eventually the relevance list will be re-ranked. The similarity between tags is one of the key factors in random walk approach.

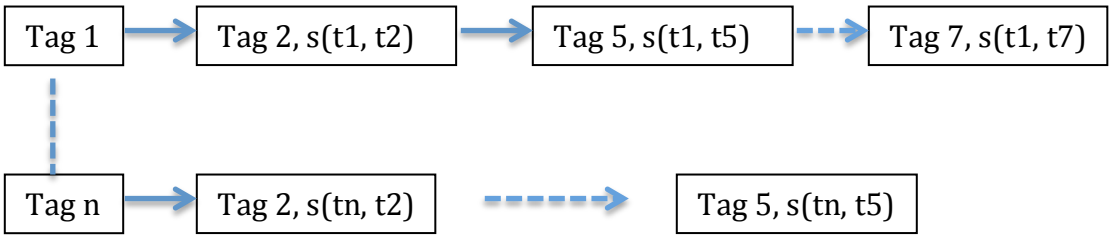


Figure 14 similarity postings

4.2 Random walk

To understand the random walk process, let's take a look at the Markov chain first. The Markov chain is a random process, and it represents the transition from one state to another among a finite numbers of possible states. The Markov chain is memory less, which means that the next state depends only on the current state.

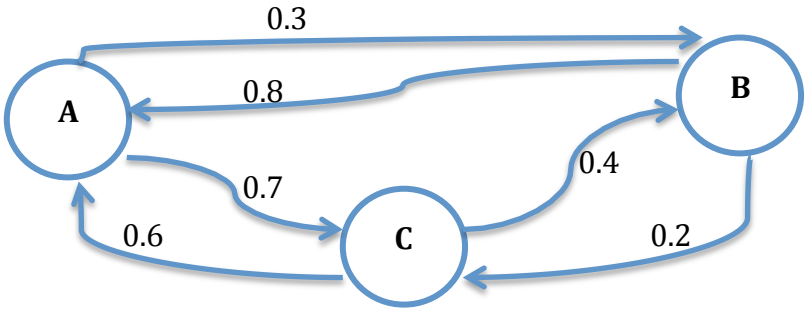


Figure 15 markov chain

Figure 15 is an example of the Markov chain. A, B and C are three states, the probability that state A transit to state B is 0.3, and state A to state C is 0.7, etc. The probability that one state transits to other states should sum up to 1. State A has two in-links with the probability 0.8 and 0.6, and state B has two in-links with the probability 0.3 and 0.4. Therefore, state B and state C will be more possible to transit to state A, while state A and state C are less possible to jump to state B.

Now look at another example in figure 16. There are three good friends, Mary, Lucy and Jack, each person has a basket. At the beginning, each person has 3 apples in their basket. The number on the edge is the probability that they visit other people. Each time one person visit another, they will buy an apple on the way, and put it into his/her friend's basket when they arrive. After a long time, Mary's basket will have more apples than Lucy and Jack, and Lucy's basket will have more apples than Jack, because Mary has the highest possibility to be visited, while Jack has the lowest.

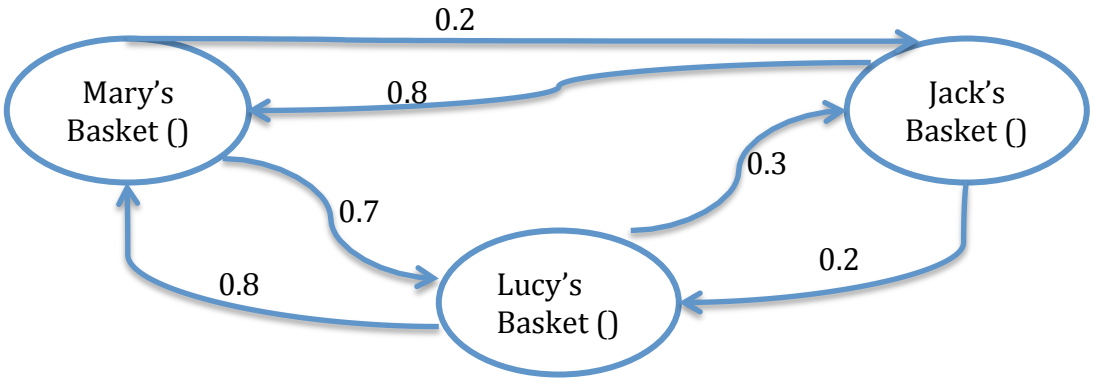


Figure 16 random walk example

Since the goal is to re-rank the relevance list, the naïve "basket-apple" model illustrated above seems to be able to increase and lower the relevance score if we replace the vertex in the graph with relevance scores. Intuitively, the edges should be replaced by similarity. The similarities, which are normalized on the out-going edges of each vertex, can be used as probabilities to guide the random walk process. The tags with high similarity score have high probability in the random walk graph, and are more possible to transit states to each other. Tags with low similarity score, on the contrary, are not that easy to jump from one state to another. If the weights are added to the destination vertex at each transition, the vertex with very high

probability in-links will accumulate more and more weights. After N iterations, vertex with high probability in-links would have its weight increased while vertex with low probability in-links would not have that much increase according to the "basket-apple" model. "Basket-apple" model is the simplest model in order to explain how random walk works. We will discuss the random walk refinement mathematically later. Figure 17 shows the expected goal the random walk should achieve. Tags with high probability edges would increase their relevance score. In the actual random walk model, the tags with very high initial relevance score will have their relevance score decreased even if they have high probability edges, because they "give" some of their weight to others. In the "basket-apple" model, vertex never loses weight, but in the actual model, vertex (tags) loses weight.

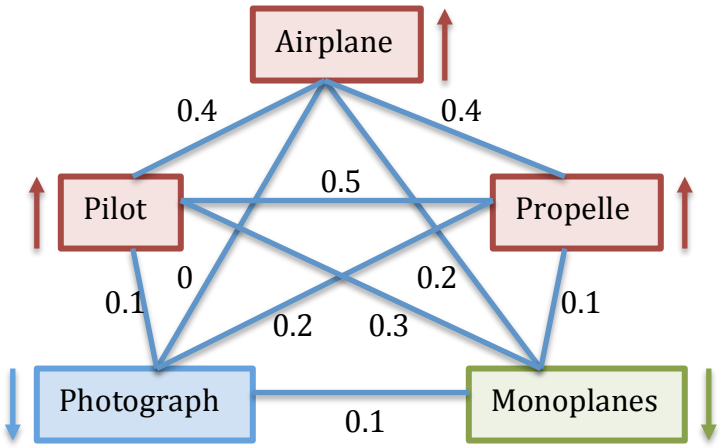


Figure 17 random walk with similarity

Now we analyze the random walk process in mathematics. We define the transition matrix **P** first.

$$p_{ij} = \frac{s_{ij}}{\sum_k s_{ik}} \quad (4.5)$$

s_{ij} is the similarity between tag i and tag j. The equation normalizes the similarity score and puts it into the matrix. Then the normalized similarity will be treated as the transition probability. The random walk refinement is a process that let the tags keep jumping from vertex to vertex by probability and redistribute the relevance importance. This process will eventually converge to some point, so that the relevance score will reach a final value.

$$r_k(i) = \alpha \sum_j r_{k-1}(j)p_{ij} + (1-\alpha)v_i \quad (4.6)$$

v_i is the initial relevance score, and $r_k(i)$ is the relevance score of node i at iteration k. Matrix representation is as follows.

$$\mathbf{r}_k = \alpha \mathbf{P}\mathbf{r}_{k-1} + (1-\alpha)\mathbf{v} \quad (4.7)$$

This process converges at some point and could be proved [3]. Assume the process terminates at n's iteration, and converges at $\mathbf{r}_\pi$. Because $0 < \alpha < 1$, there exist $\gamma < 1$, such that $\alpha < \gamma$, and we can rewrite equation (4.7).

$$\mathbf{r}_\pi = \lim_{n\to\infty} (\alpha\mathbf{P})^n \mathbf{r}_0 + (1-\alpha)\left(\sum_{i=1}^{n} \alpha\mathbf{P}^{i-1}\right)\mathbf{v} \quad (4.8)$$

$$\sum_j (\alpha\mathbf{P})_{ij}^n = \sum_j \sum_k (\alpha\mathbf{P})_{ik}^{n-1} (\alpha\mathbf{P})_{kj}$$

$$= \sum_k (\alpha\mathbf{P})_{ik}^{n-1} (\alpha \sum_j \mathbf{P}_{kj})$$

$$= \sum_k (\alpha\mathbf{P})_{ik}^{n-1} (\alpha)$$

$$\leq \sum_k (\alpha\mathbf{P})_{ik}^{n-1} (\gamma) \leq \gamma^n$$

So $(\alpha\mathbf{P})^n$ converges to zero, and therefore we can rewrite equation (4.8).

$$\mathbf{r}_\pi = (1-\alpha)\left(\sum_{i=1}^{n} \alpha \mathbf{P}^{i-1}\right)\mathbf{v} \quad (4.9)$$

$$(\mathbf{I}-\alpha\mathbf{P})\left(\sum_{i=1}^{n}(\alpha\mathbf{P})^{i-1}\right) = (\mathbf{I}-\alpha\mathbf{P})(\mathbf{I}+\alpha\mathbf{P}+(\alpha\mathbf{P})^2+\cdots+(\alpha\mathbf{P})^{n-1})$$

$$= \mathbf{I}+\alpha\mathbf{P}+(\alpha\mathbf{P})^2+\cdots+(\alpha\mathbf{P})^{n-1}-[\alpha\mathbf{P}+(\alpha\mathbf{P})^2+\cdots+(\alpha\mathbf{P})^n]$$

$$= \mathbf{I}-(\alpha\mathbf{P})^n = \mathbf{I}-0 = \mathbf{I}$$

Thus, we get:

$$(\mathbf{I}-\alpha\mathbf{P})\left(\sum_{i=1}^{n}(\alpha\mathbf{P})^{i-1}\right) = \mathbf{I} = (\mathbf{I}-\alpha\mathbf{P})(\mathbf{I}-\alpha\mathbf{P})^{-1}$$

Then:

$$\left(\sum_{i=1}^{n}(\alpha\mathbf{P})^{i-1}\right) = (\mathbf{I}-\alpha\mathbf{P})^{-1}$$

Re-write equation (4.9), we have derived final equation after convergence.

$$\mathbf{r}_\pi = (1-\alpha)(\mathbf{I}-\alpha\mathbf{P})^{-1}\mathbf{v} \quad (4.10)$$

Figure 18 gives an example of random walk refinement.

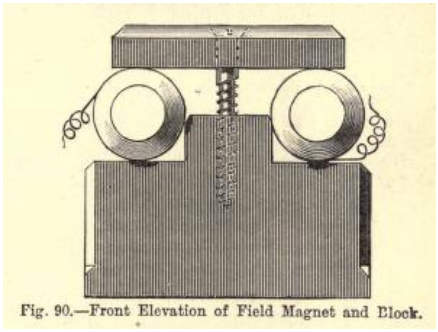


Fig. 90.—Front Elevation of Field Magnet and Block.

average precision before random walk 0.22602339181286552	average precision after random walk 0.315079365079365
armatur	armatur
magnet	**magnet**
wind	wind
wire	wire
fig	fig
wide	**coil**
middl	field
coil	coat
paper	brunswick
field	wide
coat	paper
brunswick	middl
long	electr
work	**motor**
great	layer
turn	**iron**
differ	file
short	**copper**
iron	turn
leav	smooth

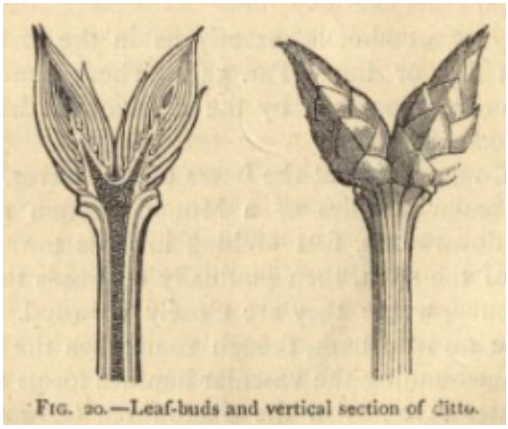


FIG. 20.—Leaf-buds and vertical section of ditto.

buds	**buds**
branches	branches
terminal	**stem**
stem	terminal
wood	spring
formed	trees
nds	**plants**
plants	**stems**
described	bark
angle	autumn
covered	**leaf**
complete	remain
vertical	dormant
spring	ends
arrangement	wet
cold	covered
trees	union
leaf	characteristic
ix	cold
remain	described
characteristic	elm
union	protected
continuous	pith
wet	hairs
bark	par
hairs	monocotyledons
increase	axillary
scales	primers
science	science
bundles	formed
par	complete
protected	leafy
autumn	increase
stems	vertical

Figure 18 random walk refinement example

Terms in bold are tags that I have picked as the best descriptive words, and we can see the positions of these tags in the relevance ranking list move up, which means that the random walk refinement is effective.

CHAPTER 5

INDEXER DESIGN AND LARGE SCALE COMPUTING

5.1 Indexer design

The book image search engine is built on Galago and Maiter cloud computing framework. These frameworks take advantage of the TupleFlow framework and asynchronous accumulative update model [6, 21]. The book image search engine implements its own indexer, storage structure, and evaluation system. Figure 19 and 20 illustrate the five stages of the image indexer. Due to the large amount of data that cannot fit into memory, we use TupleFlow stage for streaming memory. After each stage, data is written into the local hard disk, and then the next stage reads it from the local disk.

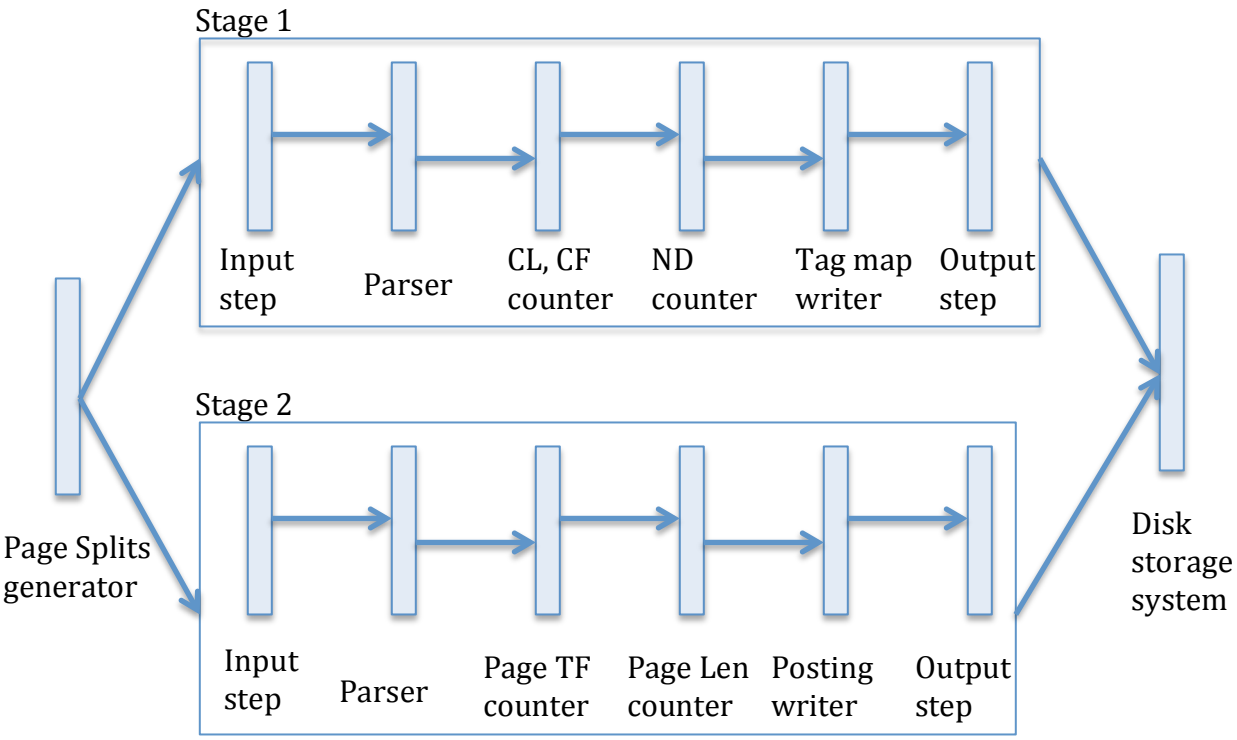


Figure 19 high level stages

In the probabilistic model, we need to know the collection length and the tag collection frequency for the smoothing model. In the similarity model, we need to know the total number of image pages. Stage 1 counts the collection length, the collection frequency and the total number of image pages. Stage 1 also creates a tag map to store tags and references to the list of image postings and similarity graph postings. Stage 2 processes every image page, and records the local parameters such as tag frequency and page length. Stage 2 also creates a posting list for each tag.

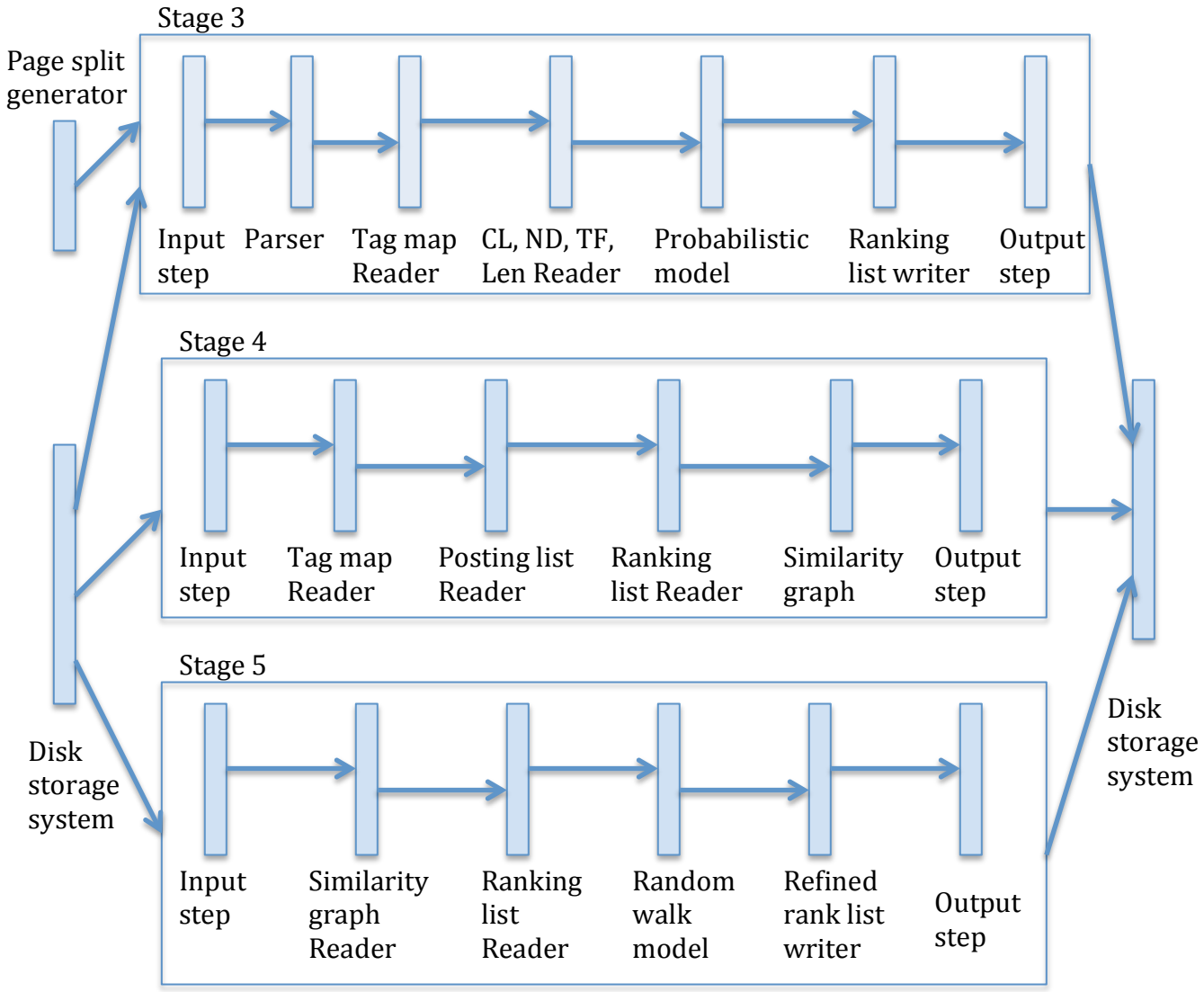


Figure 20 high level stages

Stage 3 implements a probabilistic model and creates an initial ranking list. Tag map and parameters created in stage 1 and 2 are required for stage 3, so stage 3 reads the information from the local disk first, and then applies probabilistic model to each image page. Stage 4 implements the similarity graph model and creates the similarity graph postings. Stage 5 implements the random walk model and creates a refined ranking list. Both the new ranking list and the old one are stored, because in the evaluation process, we compare the mean average precision before and after the random walk process to evaluate the effectiveness of the random walk refinement.

After describing the data flow modules and stages, I switch to the detailed indexer design and storage system design based on the five stages. First, the books are stored as TEI files (similar to xml format), so the image page parser can filter out image pages and emit a word stream for further processing.

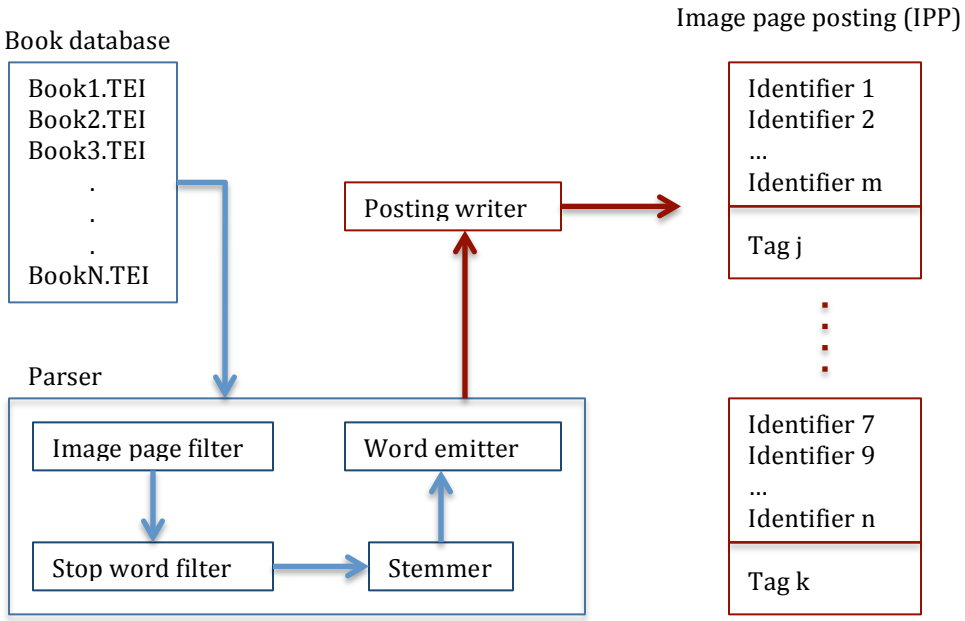


Figure 21 stage 2 design

Figure 21 illustrates how stage 2 works. Books flow into image parser, which consists of an image page filter, a stop word filter, a stemmer, and a word emitter. The posting writer creates the posting list for each tag (word). The image page posting is stored in the structure showed by figure 21. The posting writer creates a file for each tag, and uses the filename as the tag word. Now we come back to stage 1. Stage 1 creates the Tag Map, which stores the mapping information between tags and their posting list file paths. The collection frequency of each tag can also be found in the Tag Map. Figure 22 shows the design of stage 1.

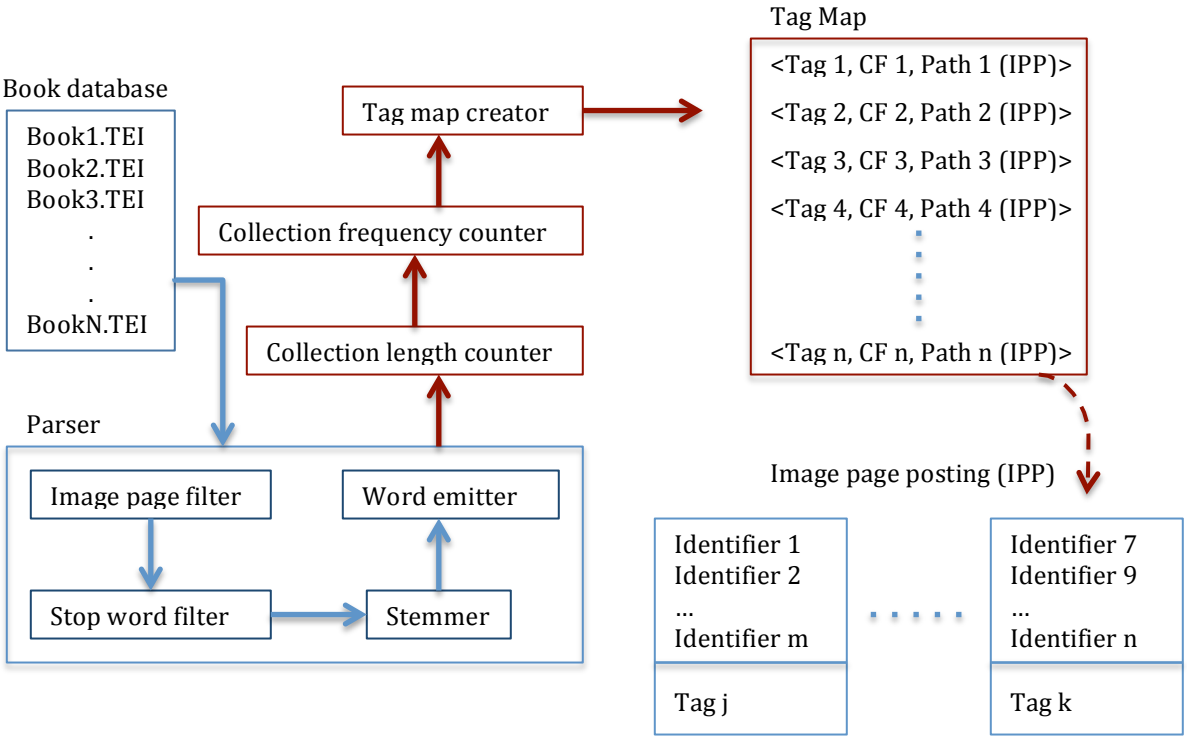


Figure 22 stage 1 design

Tag Map is stored as the HashMap object, and it is loaded into the memory when the image search engine server starts. Stage 3 applies the probabilistic model to image pages and creates tag-ranking list for each image page. The tag-ranking

lists are stored as HashMap objects for fast access purpose, and every tag-ranking file uses the page identifier as the file name. Figure 23 illustrates stage 3.

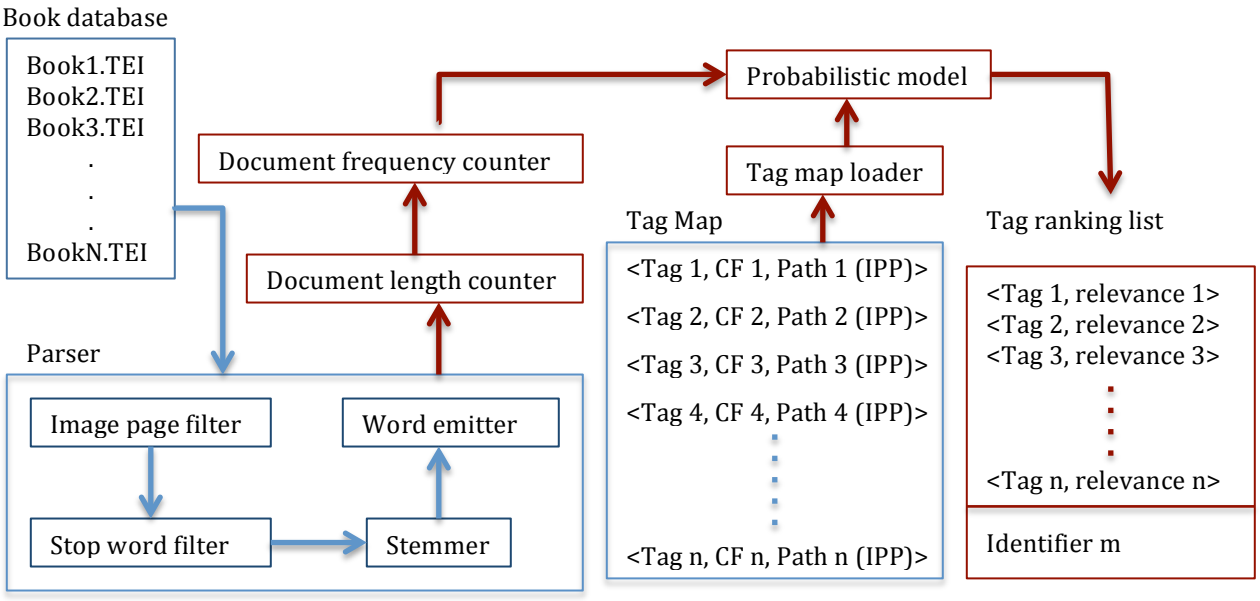


Figure 23 stage 3 design

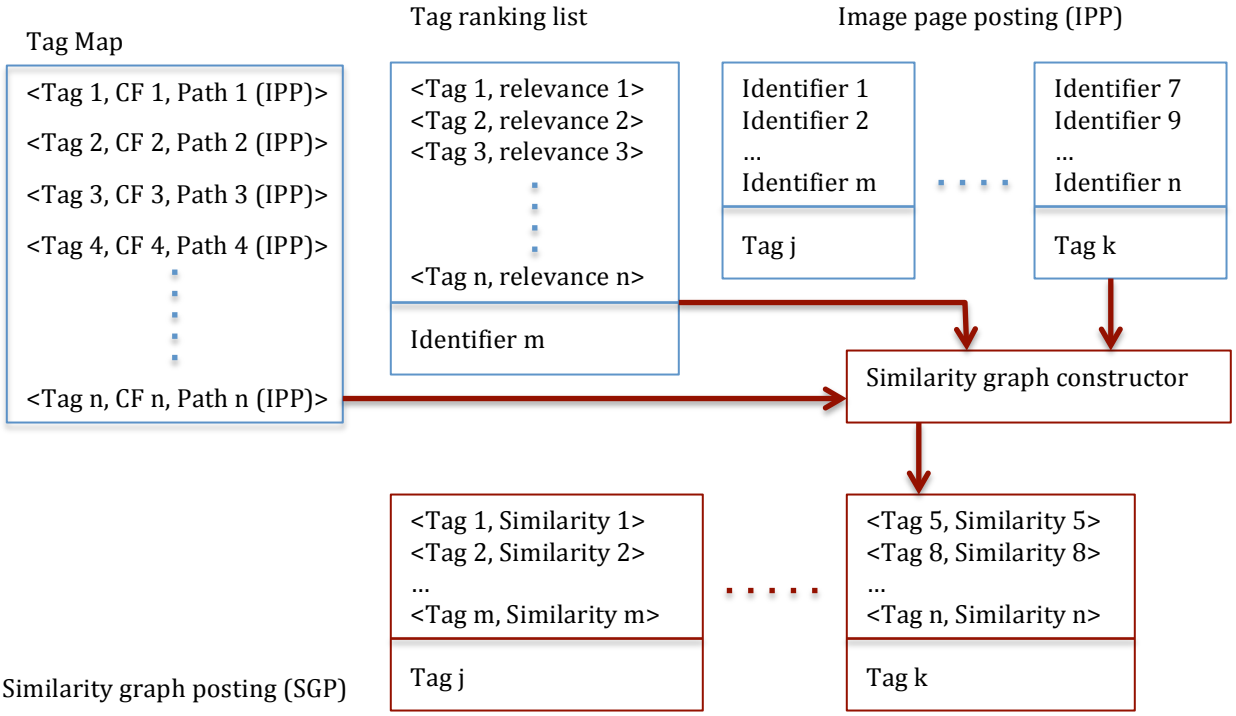


Figure 24 stage 4 design

Stage 4 creates the similarity graph for the whole collection. The similarity graph is represented by the adjacency list format. Each tag has its own similarity adjacency list file, and as usual, the file is named after the tag. Figure 24 illustrates the graph construction process and the storage data structure.

Stage 5 performs the random walk refinement on the tag-ranking list of each image page, and creates refined tag-ranking list. Figure 25 shows the random walk refinement process. The refined tag-ranking list uses the same storage method and data structure as the original one.

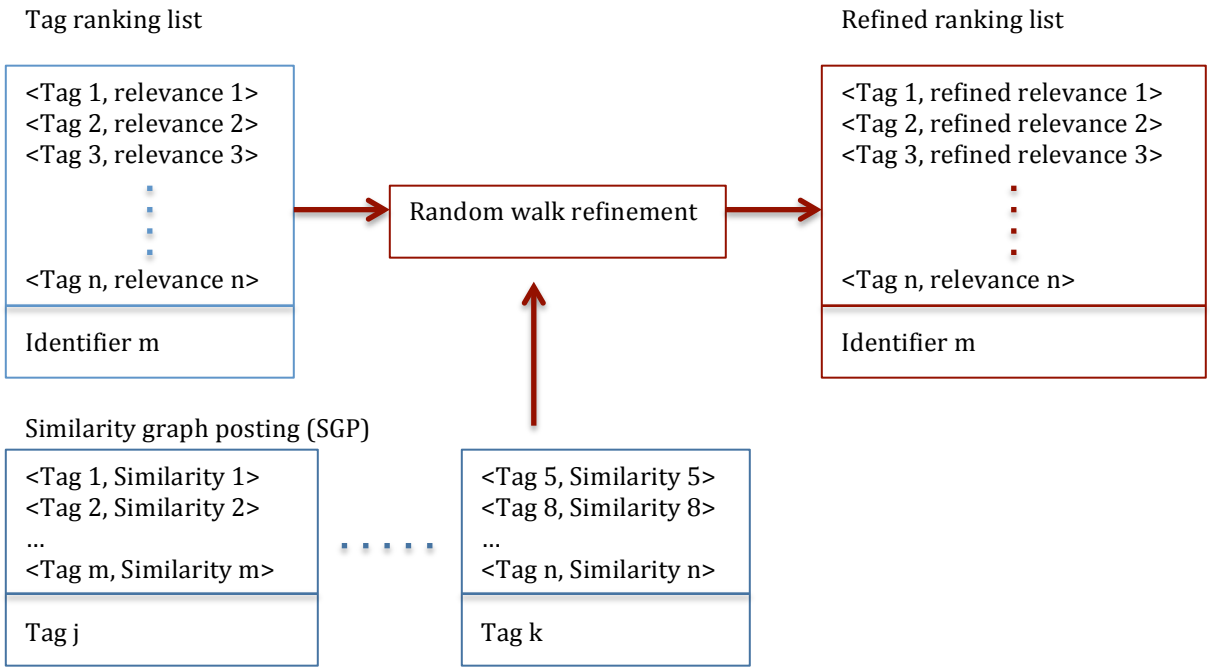


Figure 25 stage 5 design

The tag map, the image page posting, and the refined ranking list forms the final index for the image search. The search engine looks up the tag map for the query first, then locates the query words in the posting list and finds out all the pages that contain query words. Next, the search engine pulls out the relevance score of the query words in the page's refined ranking list. Finally, the search engine

ranks the pages by combining the relevance score and return the results to the interface. Figure 26 illustrates how the image search engine works.

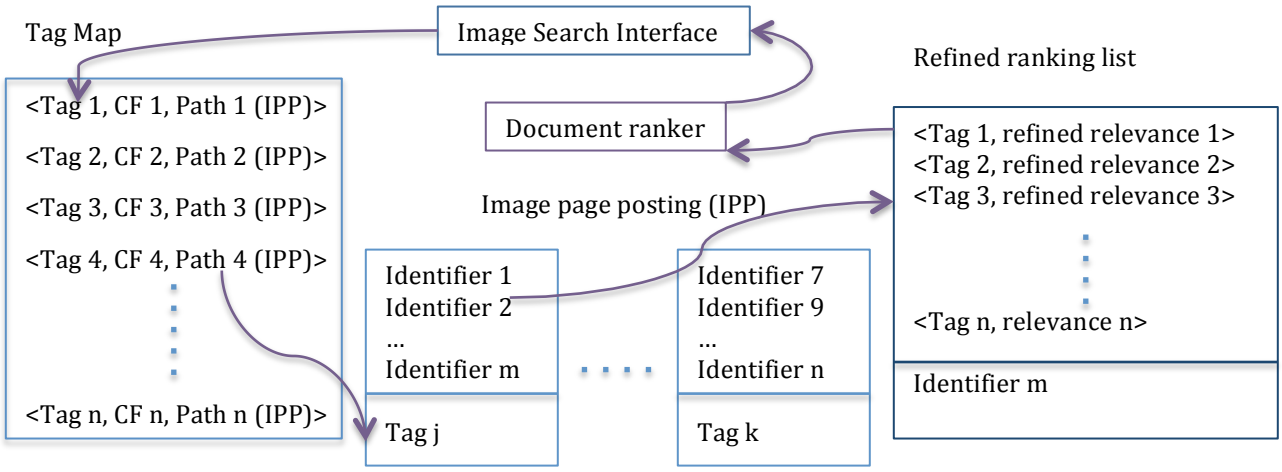


Figure 26 query search

5.2 Indexing efficiency analysis

In this thesis, I use a relatively small prototype, which has 4736 image pages. The whole index process lasts for 3.8 hours, and the random walk refinement process needs 3.2 hours, which takes up 84% of the entire execution time. The following part in this section demonstrates the experiment of memory usage and execution time according to five implementation stages.

Experiment summary

Number of books indexed	50
Number of pages indexed	4736
Machine CPU	Dual-core 2.0GHz
Machine memory	2 GB
Number of machines	1

Table 1 efficiency analysis summary

Execution time and memory usage

STAGE ID	STEP NAME	MEMORY USAGE	MEMORY OCCUPANCY	EXECUTION TIME
1	Counting collection length	0.6 GB	31.2%	125 seconds
2	Collecting identifier list for each tag	0.5 GB	22.5%	167 seconds
3	Apply probabilistic model and create ranked tag list	0.5 GB	23%	101 seconds
4	Similarity-graph construction	0.2 GB	10.2%	1720 seconds (29) minutes
5	Random-walk refinement	0.2 GB	8.5%	11500 seconds (192) minutes

Table 2 execution time and memory usage

Disk usage

Storage module name	Disk usage	Description
Frequency map	2.3 MB	Mapping between tags and posting list references
Image page list	138 MB	Mapping between tag T and a list of pages that contain tag T
Document tag list	19 MB	Mapping between page D and tag list of page D
Refined ranking list	19 MB	Mapping between page D and refined tag list of page D
Similarity graph	332 MB	Graph with tags as it's vertices and similarity as it's edges
Similarity temporary file	192 MB	Intermediate result

Table 3 disk usage

Table 1 introduces the experiment environment. Table 2 summarizes the run time and memory usage. Table 3 shows the disk usage and brief description of each storage module. The detailed experiment plots are in the appendix.

The total storage needed for indexing is 706 MB, and the total run time is 227 minutes. The random walk refinement process takes up 84% of the entire run time and is the bottleneck for fast indexing.

5.3 Cloud computing

The prototype uses a relatively small dataset that has 50 books. Assume that each book has the same number of picture page and each page contain the same number of word, we need $\frac{192\times\frac{5,000,000}{50}}{60\times24\times365} = 36$ years, which is incredibly long, to finish the random walk refinement for 5 million books. The cloud computing model, at this point, is proposed to solve the efficiency problem and scale issue.

The emergence of the cloud computing is inevitable. Since Internet service grew up rapidly during 1990s [14], it becomes as important as electricity in daily life. Some Internet services such as search engines and social networks are processing a mass amount of data everyday. For example, Facebook has over 900 million user profiles [15] and LinkedIn stores more than 150 million user profiles [16]. Massive computing becomes indispensable in the 21st century, therefore, the concept "computing as a utility" has been proposed [17]. In the following chapters, we talk about how the cloud computing models are designed to solve the image search scale issue.

CHAPTER 6

MAPREDUCE MODEL AND MAITER MODEL

6.1 Introduction

6.1.1 MapReduce

The book image search project consists of jobs that need to process large amount of data, such as inverted indices generation, similarity graph construction and random walk iteration. The inputs are always large and it would take forever if only a single machine is used. The cloud computing models make it possible to distribute large-scale jobs to clusters. In this section, we introduce the MapReduce model for parallel computing. The MapReduce model is well designed and has a neat interface. Users just need to express the computation they want to perform and the framework will take care of all the parallelization, data distribution and fault tolerant [9].

MapReduce, similar to functional programming in some aspect, has a user defined “map” function and “reduce” function to process key-value pairs. Input key-value pairs flow into a “map”, and intermediate key-value pairs will be created according to the implementation of the “map” function. Then values associated with the same key will be collected by the “reduce” function, and now it’s up to the user on how to play with these key-list pairs. Standing on user’s position, what needs to be done is just code up the “map” function and “reduce” function, as well as setting up the very straightforward job configuration. Figure 27 illustrates the MapReduce model from the user’s point of view.

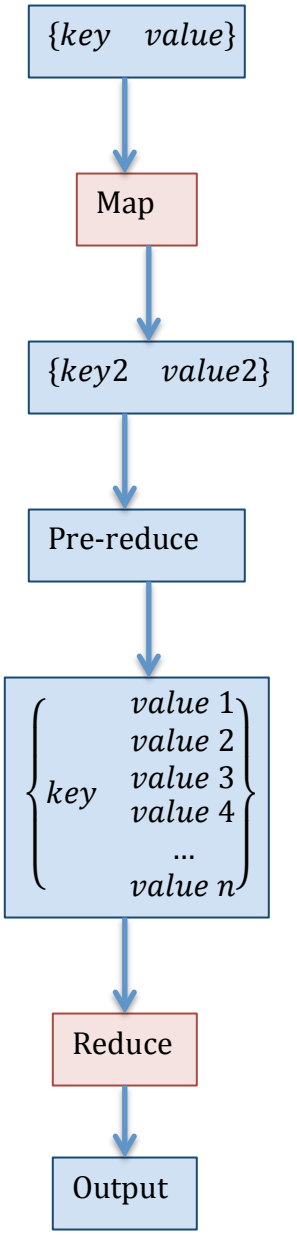


Figure 27 MapReduce flow

There are many different implementations of the MapReduce model. In this thesis, all the MapReduce experiments are conducted on the Apache Hadoop framework. I discuss the architecture of Hadoop framework and how it works in the appendix. Even the comparison experiment between the MapReduce and Maiter model, which is discussed in the following section, gives up the MapReduce model for the random walk refinement, it is still a good option to deploy other jobs of

image searching such as document parsing and similarity graph construction on Hadoop framework.

6.1.2 Maiter

Let me reiterate our goals for introducing the cloud computing models: make the random walk refinement faster, which takes up to 84% of the entire running time. Recall from section 4.2, the math representation of random walk is as follows.

$$\mathbf{r}_k = \alpha \mathbf{P}\mathbf{r}_{k-1} + (1-\alpha)\mathbf{v}$$

In this equation, $\mathbf{r}_{k-1}$ is the relevance score vector at (k-1) iteration, and $\mathbf{r}_k$ is the relevance score vector at k iteration. Clearly, it is an iterative process. In recent networking technologies and online services, huge amount of data is collected everyday, i.e. as I mentioned in section 5.2, Facebook and LinkedIn already have hundreds of millions of registered users, and a bunch of data mining or machine learning algorithms are applied to their collected data such as PageRank [20]. These algorithms usually require an iterative process. Cloud computing models such as MapReduce, which I discussed in section 6.1, are used to accelerate the iterative process. Take my MapReduce experiment for example, each iteration, the job will be split into many tasks and distributed to a bunch of machines for processing, and the result will be written on the file system. The next iteration will read the result of the previous iteration as the input. In this distributed computing approach, the next iteration cannot start before the current iteration has completed, and the current iteration is base on the result of the completed result of the previous one.

In the following section, we introduce a model that is different from the MapReduce model in two places. First, the current iteration does not rely on the completed result of the previous iteration. On the contrary, it accepts every piece of update from any iteration (both previous and current), and we call this mechanism "accumulative update". Second, the next iteration does not need to wait for the current iteration to complete. Iterations can start as long as it receives updates (it is possible that the first iteration and the last iteration are running at the same time), and we call it "asynchronous update" [21].

6.2 Random walk with Maiter model

6.2.1 Asynchronous accumulative update

Let me begin with the simplest case. Assume that we have a graph of n nodes, and at iteration k, the value of node j equals to the summation of the values of other nodes at iteration k-1 plus a constant c.

$$v_j^k = v_1^{k-1} + v_2^{k-1} + \cdots + v_{j-1}^{k-1} + v_{j+1}^{k-1} + \cdots + v_n^{k-1} + c \quad (6.1)$$

Assume that the value of node j at iteration k could also be derived by adding up the value of node j at iteration k-1 and the update factor Δv_j^k of node j from iteration k-1 to iteration k [21].

$$v_j^k = v_j^{k-1} + \Delta v_j^k \quad (6.2)$$

In equation (6.2), Δv_j^k is the update factor of node j from iteration k-1 to iteration k. Plug (6.2) in equation (6.1), and we get equation (6.3).

$$v_j^k = \left(v_1^{k-2} + \Delta v_1^{k-1}\right) + \left(v_2^{k-2} + \Delta v_2^{k-1}\right) + \cdots + \left(v_{j-1}^{k-2} + \Delta v_{j-1}^{k-1}\right) +$$

$$\left(v_{j+1}^{k-2} + \Delta v_{j+1}^{k-1}\right) + \cdots + \left(v_n^{k-2} + \Delta v_n^{k-1}\right) + c \quad (6.3)$$

According to liner operation's commutative and associative property, we can rewrite equation (6.3) like the follows.

$$v_j^k = \left(v_1^{k-2} + v_2^{k-2} + \cdots + v_{j-1}^{k-2} + v_{j+1}^{k-2} + \cdots + v_n^{k-2}\right) + c +$$

$$\left(\Delta v_1^{k-1} + \Delta v_2^{k-1} + \cdots + \Delta v_{j-1}^{k-1} + \Delta v_{j+1}^{k-1} + \cdots + \Delta v_n^{k-1}\right) \quad (6.4)$$

Notice that $\left(v_1^{k-2} + v_2^{k-2} + \cdots + v_{j-1}^{k-2} + v_{j+1}^{k-2} + \cdots + v_n^{k-2}\right) + c = v_j^{k-1}$, thus we replace the first part in equation (6.4) with v_j^{k-1} and get (6.5).

$$v_j^k = v_j^{k-1} + \left(\Delta v_1^{k-1} + \Delta v_2^{k-1} + \cdots + \Delta v_{j-1}^{k-1} + \Delta v_{j+1}^{k-1} + \cdots + \Delta v_n^{k-1}\right) \quad (6.5)$$

Compare equation (6.2) and (6.5), we have derived the expression for update factor [21].

$$\Delta v_j^k = \left(\Delta v_1^{k-1} + \Delta v_2^{k-1} + \cdots + \Delta v_{j-1}^{k-1} + \Delta v_{j+1}^{k-1} + \cdots + \Delta v_n^{k-1}\right) \quad (6.6)$$

$$\Delta v_j^k = \sum_{i \in N, i \neq j} \Delta v_i^{k-1} \quad (6.7)$$

Equation (6.6) and (6.7) are expressions for update factor of node j at iteration k. To describe how accumulative update work, we expand equation (6.7) first.

$$\Delta v_j^k = \sum_{i \in N, i \neq j} \Delta v_i^{k-1} = \sum_{i \in N, i \neq j} \sum_{t \in N, t \neq i} \Delta v_t^{k-2} = \sum_{i \in N, i \neq j} \ldots \sum_{r \in N} \Delta v_r^1 \quad (6.8)$$

In equation (6.8), Δv_r^1 is the initial update factor, which depends on the initial value of each node. We can see that equation is a linear operation, and thus, the update factors accumulate all the way up to iteration k.

Equation (6.8) could be represented with a tree structure, and Figure 22 illustrates this tree-like structure. Assume the bottom level of this tree is level 1, and

it corresponds to the first iteration. The leaf nodes are the initial update factors, and the root is Δv_j^k.

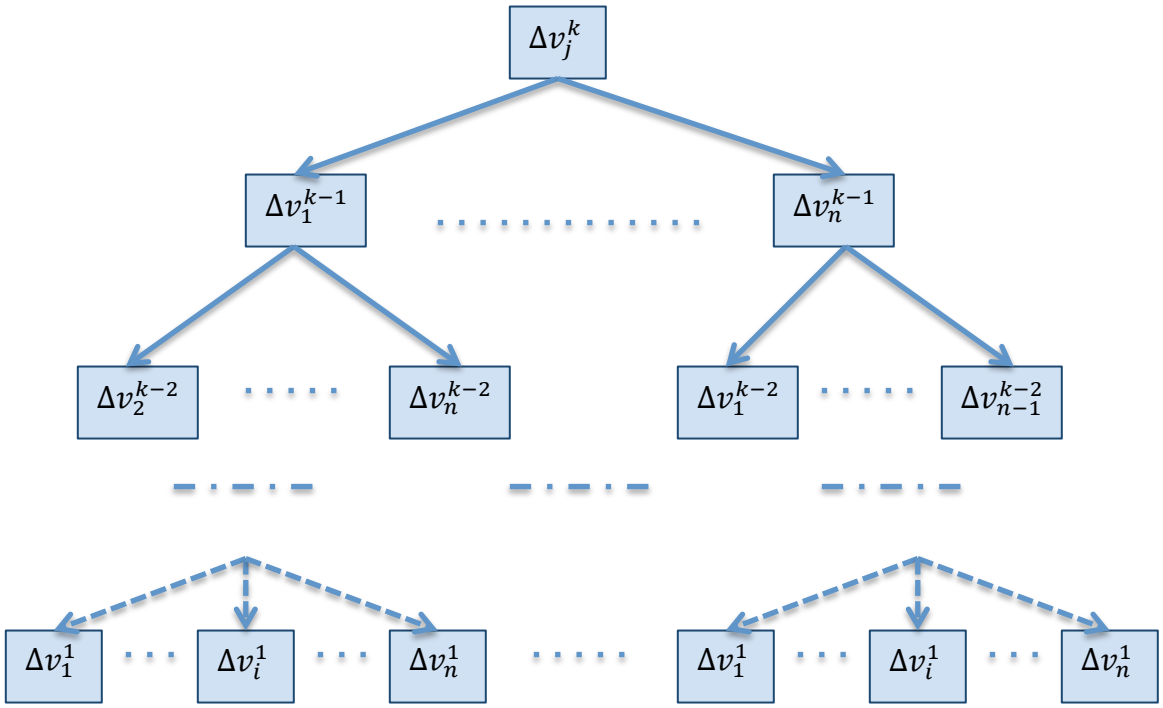


Figure 28 accumulative update

In Figure 22, replace each node with a "+" operator from level 2 up to the top, we can easily expand equation (6.8) to the following polynomial.

$$\Delta v_j^k = c_1 \cdot \Delta v_1^1 + c_2 \cdot \Delta v_2^1 + \cdots + c_j \cdot \Delta v_j^1 + \cdots + c_n \cdot \Delta v_n^1 \quad (6.9)$$

In equation (6.9), $c_1 = c_2 = \cdots = c_n = (n-1)^{k-2}$. This series of constant c is corresponding to the simplest case which is described in equation (6.1) and (6.6). For complicated iteration representation, $c_1 \ldots c_n$ may not equal, but they must be all constants as long as the iteration representation is linear. Till now, we have proved that accumulative update is possible and we have derived the update factor in equation (6.7). Now I would like to prove that it could be performed asynchronously in next paragraph.

Pick up a node randomly from the tree in Figure 23. Let's assume we have picked up node m in iteration p. Notice that there is a sub-tree with Δv_m^p as the root, and according to equation (6.9), we get the following expression for Δv_m^p.

$$\Delta v_m^p = u_1 \cdot \Delta v_1^1 + u_2 \cdot \Delta v_2^1 + \cdots + u_m \cdot \Delta v_m^1 + \cdots + u_n \cdot \Delta v_n^1 \quad (6.10)$$

In equation (6.10), $u_1 \ldots u_n$ are constants. Follow the tree structure, the value of Δv_m^p will be accumulated all the way up to the root Δv_j^k, and is part of the value of Δv_j^k.

$$\Delta v_j^k = (c_1 - u_1) \cdot \Delta v_1^1 + (c_2 - u_2) \cdot \Delta v_2^1 + \cdots + (c_n - u_n) \cdot \Delta v_n^1 + \Delta v_m^p \quad (6.11)$$

Equation (6.11) tells us that it does not matter when Δv_m^p is calculated. Therefore, we've proved that the accumulative update can be done asynchronously, and the next iteration does not need to wait for the current completion. Even through the proof is based on the simplest iterative representation described in (6.1), it is true for all linear cases. You can prove it by replacing the nodes in the tree with the proper linear expression.

6.2.2 Random walk refinement

Since our goal is to accelerate random walk refinement process, we should take advantage of this accumulative update model to design the new asynchronous accumulative updated random walk refinement.

The math to derive the random walk accumulative update expression is very similar to what we have done in section 6.2.2, but the assumptions are more general. Assume $v^k = \{v_1^k, v_2^k, \ldots, v_n^k\}$ is a vector at iteration k, we have $v^k = F(v^{k-1})$. For element j, we have update function as follows [21].

$$v_j^k = f_j\left(v_1^{k-1}, v_2^{k-1}, \dots, v_n^{k-1}\right) \quad (6.12)$$

Apply iterative update equation for element j to the random-walk process for the books.

$$R_j^k = \alpha \cdot \sum_{\{i|(i\to j)\in E\}} R_i^{k-1} \cdot \frac{S_{(i\to j)}}{\sum_{\{m|(i\to m)\in E\}} S_{(i\to m)}} + (1-\alpha)\cdot v_j \quad (6.13)$$

R_j^k is the relevance score for node j at iteration k, α is the damping factor, in the experiment, we assign 0.8 to the damping factor. $S_{(i\to j)}$ is the similarity between node i and node j and v_j is the initial score for node j.

Let $S_{ij} = \frac{S_{(i\to j)}}{\sum_{\{m|(i\to m)\in E\}} S_{(i\to m)}}$, we can rewrite equation (6.13).

$$R_j^k = \alpha \cdot \left(S_{1j}R_1^{k-1} + S_{2j}R_2^{k-1} + \cdots + S_{nj}R_n^{k-1}\right) + (1-\alpha)v_j \quad (6.14)$$

Assume that R_j^k can be obtained by adding up an accumulative update factor ΔR_j^k to the previous iteration of R_j^{k-1}, we have the following equation, which is very similar to equation (6.2).

$$R_j^k = R_j^{k-1} + \Delta R_j^k \quad (6.15)$$

Plug equation (6.15) in equation (6.14), we get the following equation.

$$R_j^k = \alpha\left[S_{1j}\left(R_1^{k-2} + \Delta R_1^{k-1}\right) + S_{2j}\left(R_2^{k-2} + \Delta R_2^{k-1}\right) + \cdots + S_{nj}(R_n^{k-2} + \Delta R_n^{k-1})\right]$$

$$+(1-\alpha)v_j \quad (6.16)$$

Rewrite equation (6.16) according to linear operation's commutative and associative property.

$$R_j^k = \alpha\left(S_{1j}R_1^{k-2} + S_{2j}R_2^{k-2} + \cdots + S_{nj}R_n^{k-2}\right)$$

$$+\alpha\left(S_{1j}\Delta R_1^{k-1} + S_{2j}\Delta R_2^{k-1} + \cdots + S_{nj}\Delta R_n^{k-1}\right) + (1-\alpha)v_j \quad (6.17)$$

Notice that $\alpha\left(S_{1j}R_1^{k-2} + S_{2j}R_2^{k-2} + \cdots + S_{nj}R_n^{k-2}\right) + (1-\alpha)v_j = R_j^{k-1}$, plug it in equation (6.17), we get simplified (6.18).

$$R_j^k = R_j^{k-1} + \alpha\left(S_{1j}\Delta R_1^{k-1} + S_{2j}\Delta R_2^{k-1} + \cdots + S_{nj}\Delta R_n^{k-1}\right) \quad (6.18)$$

Compare equation (6.15) and (6.18), we get the expression for random walk update factor.

$$\Delta R_j^k = \alpha\left(S_{1j}\Delta R_1^{k-1} + S_{2j}\Delta R_2^{k-1} + \cdots + S_{nj}\Delta R_n^{k-1}\right) \quad (6.19)$$

$$\Delta R_j^k = \alpha \cdot \sum_{\{i|(i\to j)\in E\}} \Delta R_i^{k-1} \cdot \frac{S_{(i\to j)}}{\sum_{\{m|(i\to m)\in E\}} S_{(i\to m)}} \quad (6.20)$$

We calculate the value of iteration k by adding up the value of iteration k-1 and the update factor of iteration k. Summing up the update factor of iteration k-1 will give us the update factor of iteration k. In section 6.2.2, we have already proved the correctness of asynchronous accumulative update process, and Equation (6.15) and (6.20) shows how accumulative iterative update can be applied on random-walk process [21].

6.3 Comparison between MapReduce and Maiter

6.3.1 MapReduce system design

We are using Hadoop framework as well as Hadoop Distributed File System (HDFS). I discuss the framework and HDFS in detain in the appendix. The input files have the tag node as the key, and the value string contains the relevance score and the neighbor nodes information. The temporary result will be written into HDFS, and read as input of the next iteration. Figure 23 illustrates the flow chart of the MapReduce implementation for the random walk refinement.

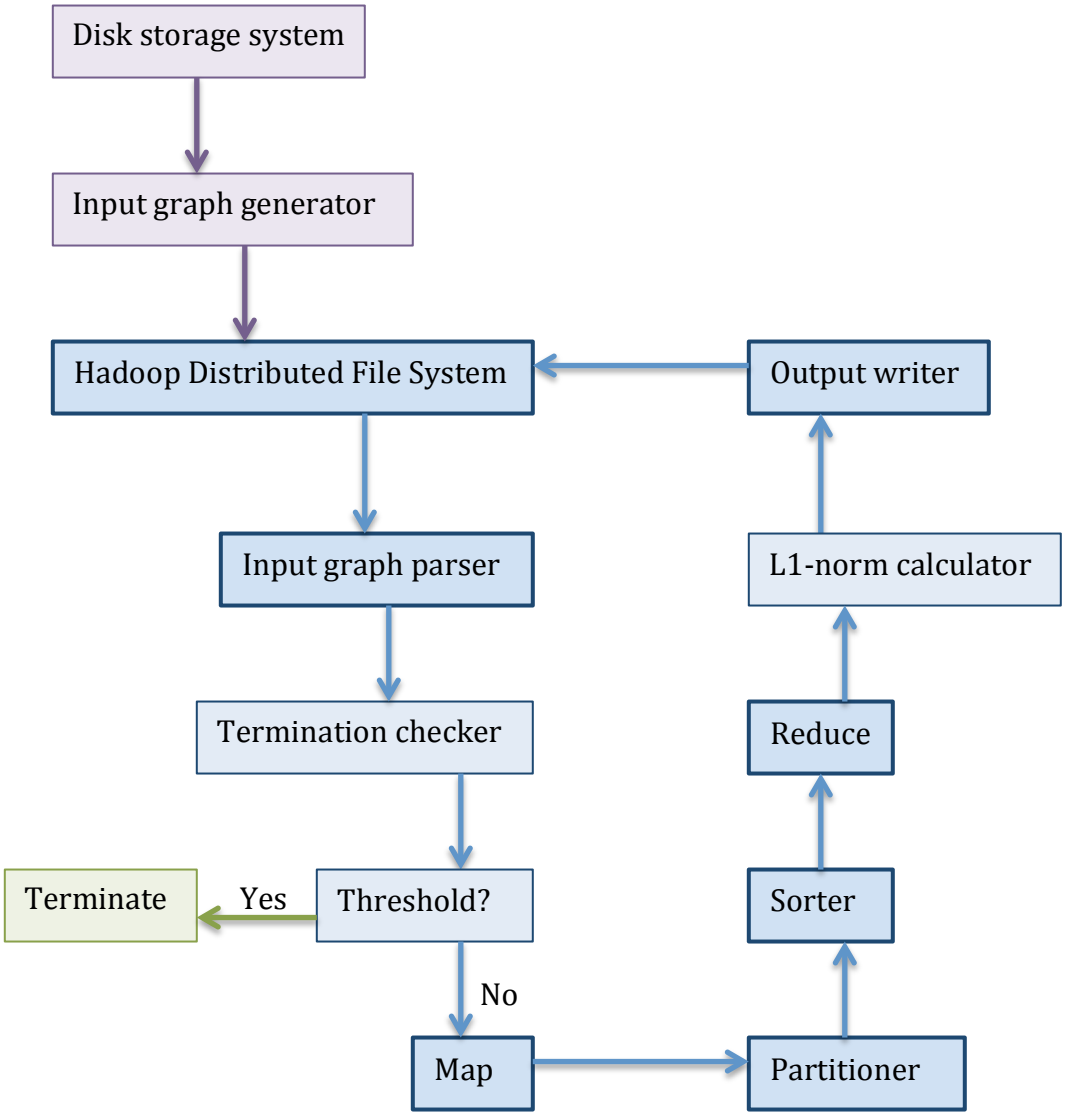


Figure 29 MapReduce implementation

MapReduce process is built on Hadoop framework. We implement the input graph generator, key-value pair parser, iteration reader, termination checker and the map function in Map step. L1-norm calculator and reduce function are implemented in Reduce step. The process in Figure 29 is iterative (The blue circle), which means it will run a bunch of times until the termination check reaches the threshold. The output will be written into HDFS, and the next iteration will read the output of the previous iteration from the HDFS. The termination checker will take

the difference of L1-norm values at current and previous iteration. If the difference meets the threshold, the program terminates, and if not, it goes into the next iteration and repeats the same process.

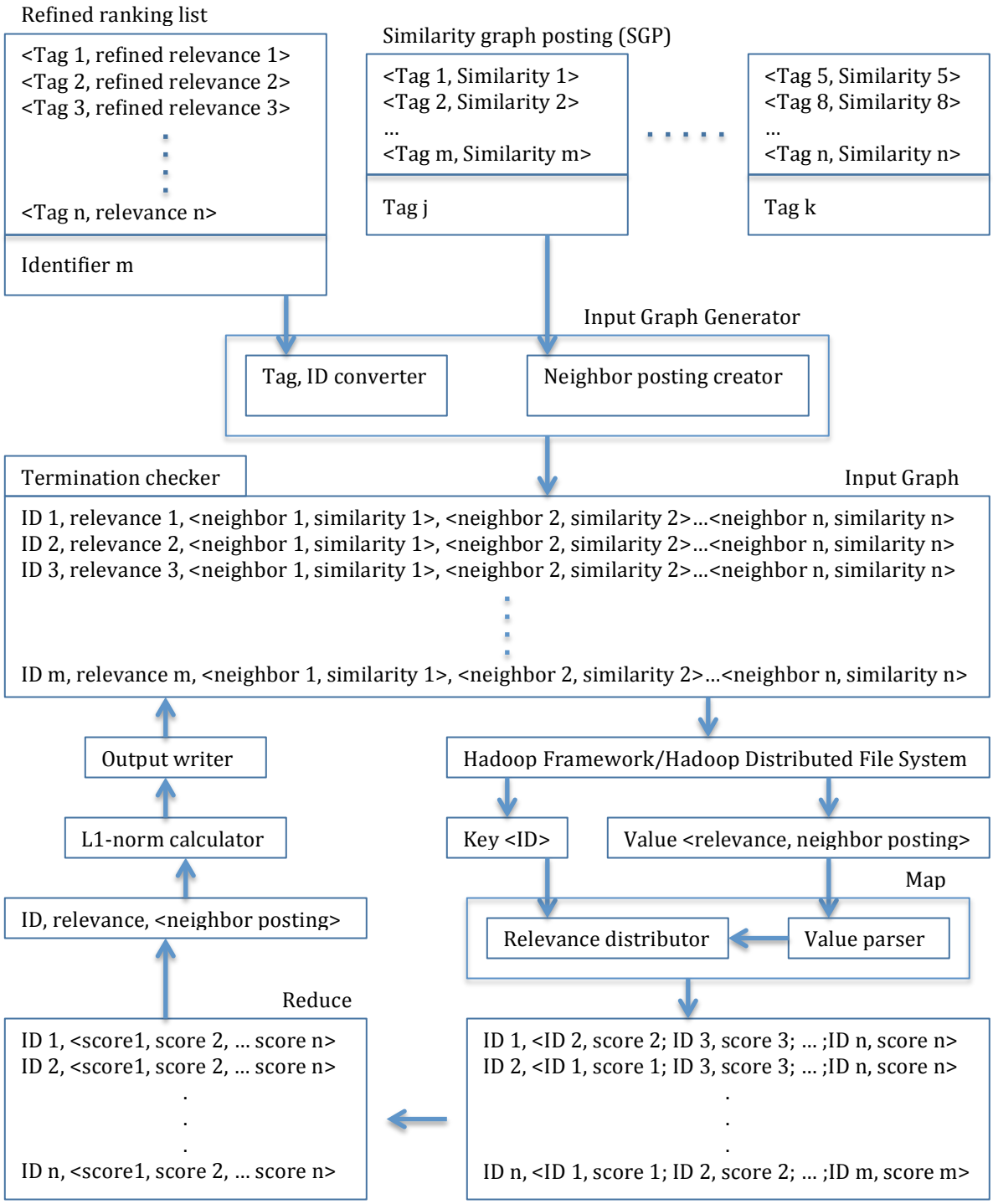


Figure 30 MapReduce design

According to my experiment, 90 iterations are need for threshold of 0.000001. Therefore, one of the bottlenecks for MapReduce is actually the I/O restriction. Intensive I/O communications with local disk slows down the computation speed a lot. The other bottleneck is synchronized mechanism. It wastes lot of time waiting the previous iteration to finish.

6.3.2 Maiter system design

In section 6.2.1 and 6.2.2, we dove into Maiter's math foundation, but here, I would like to talk about the implementation. Let's begin our discussion with math again by exploring equation (6.15) and (6.20). Expanding these equations, we get the following new expressions.

$$R_j^k = R_j^{k-1} + \Delta R_j^k = R_j^{k-2} + \Delta R_j^{k-1} + \Delta R_j^k = R_j^{k-3} + \Delta R_j^{k-2} + \Delta R_j^{k-1} + \Delta R_j^k = \cdots$$

$$= R_j^0 + \Delta R_j^1 + \Delta R_j^2 + \cdots + \Delta R_j^{k-1} + \Delta R_j^k \quad (6.21)$$

$$\Delta R_j^k = \sum_{\{i|(i\to j)\in E\}} \Delta_{ij}^{k-1} \quad (6.22)$$

Where

$$\Delta_{ij}^{k-1} = \alpha \cdot \Delta R_i^{k-1} \cdot \frac{S_{(i\to j)}}{\sum_{\{m|(i\to m)\in E\}} S_{(i\to m)}}$$

According to (6.21) and (6.22), R_j^k is linear, and therefore, we derive rules for asynchronous accumulative updates as follows.

When node j receives an update Δ_{ij}(we do not need to care about which iteration the update comes from), we do two things.

First step, we update R_j by adding up the coming updates $R_j = R_j + \Delta_{ij}$.

Second step, Δ_{ij} is part of ΔR_j, and when node j receives it, we let $\Delta_j = \Delta_{ij}$, because it has nothing to do with node i. Then we distribute Δ_j to node k with the value $\Delta_j \cdot \frac{S_{(j\to k)}}{\sum_{\{m|(j\to m)\in E\}} S_{(j\to m)}}$.

Because R_j^k is totally linear, we do not need to wait for the previous iteration to complete before the next iteration starts, we can just repeat the two steps whenever the update is available.

The Maiter model overcomes these two bottlenecks mentioned in section 6.3.1 and improves the performance significantly. The Maiter model is being built on the Maiter framework (the framework and the model share the same name), and Figure 31 illustrates how it is implemented.

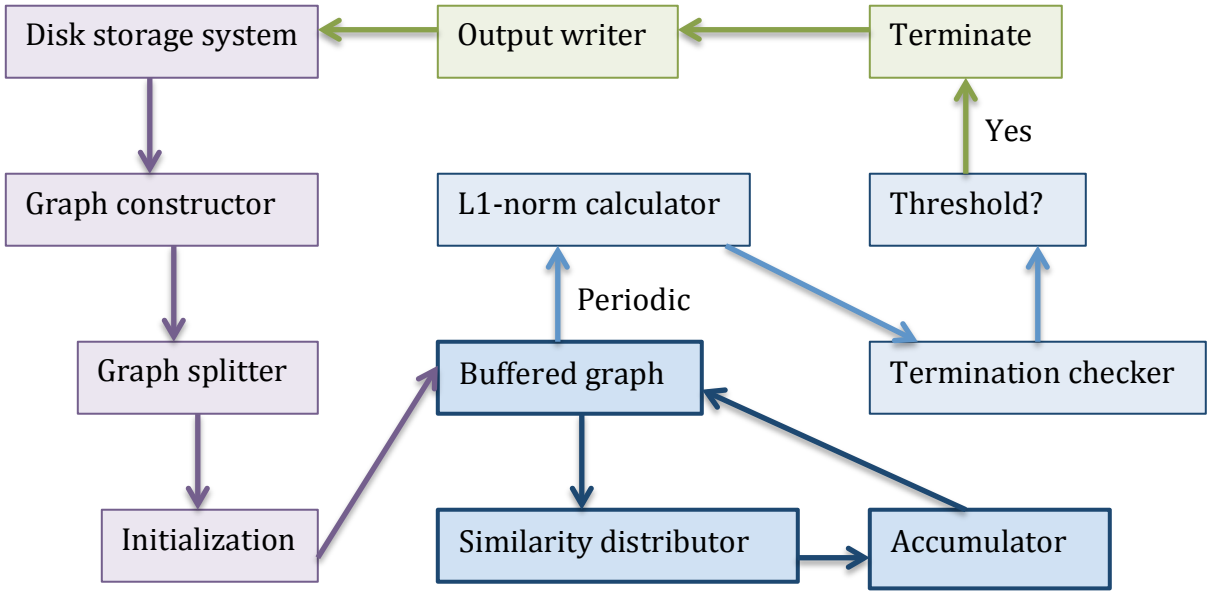


Figure 31 Maiter implementation

The Graph constructor creates a global graph described in section 7.2 for the whole collection. As I mentioned in section 7.2, this step is tricky but crucial, because it eliminates the needs for multi-loading, which is really time-consuming

when the number of a graph is large. The initialization step assigns an original relevance score to each tag node. The similarity distributer step decides how a tag similarity importance is propagated and the accumulator step performs the asynchronous accumulative updates. Termination checker compute the L1-norm difference periodically (the period is specified by users). When the L1-norm difference meets the threshold, the program enters the termination process and the result will be written into the local disk storage system.

Figure 32 and Figure 33 shows the Maiter system design. The input graph generator and the graph splitter are illustrated in Figure 32, and these two modules are crucial to the system's performance. I discuss the reason why the graph split has such big influence on the performance in section 7.2.

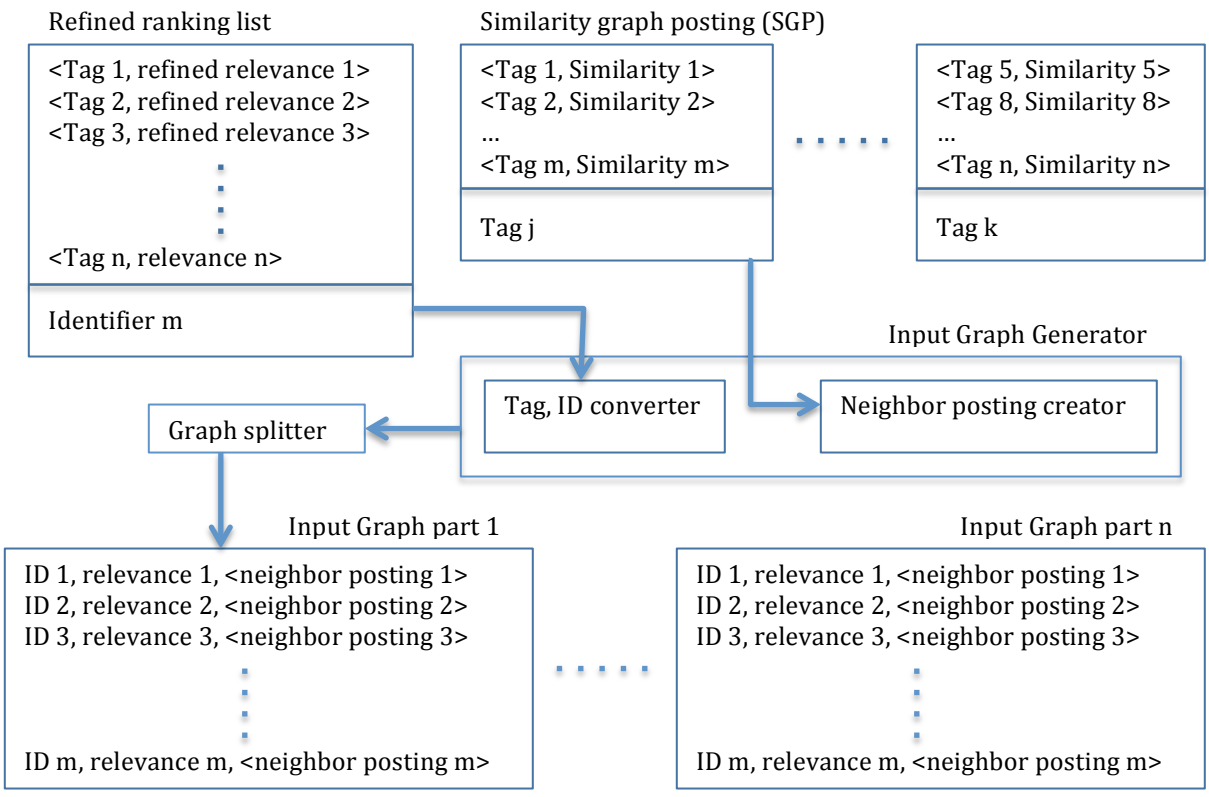


Figure 32 Maiter graph construction

In the Maiter design, the similarity distributor (relevance distributor) and the accumulator are the core modules.

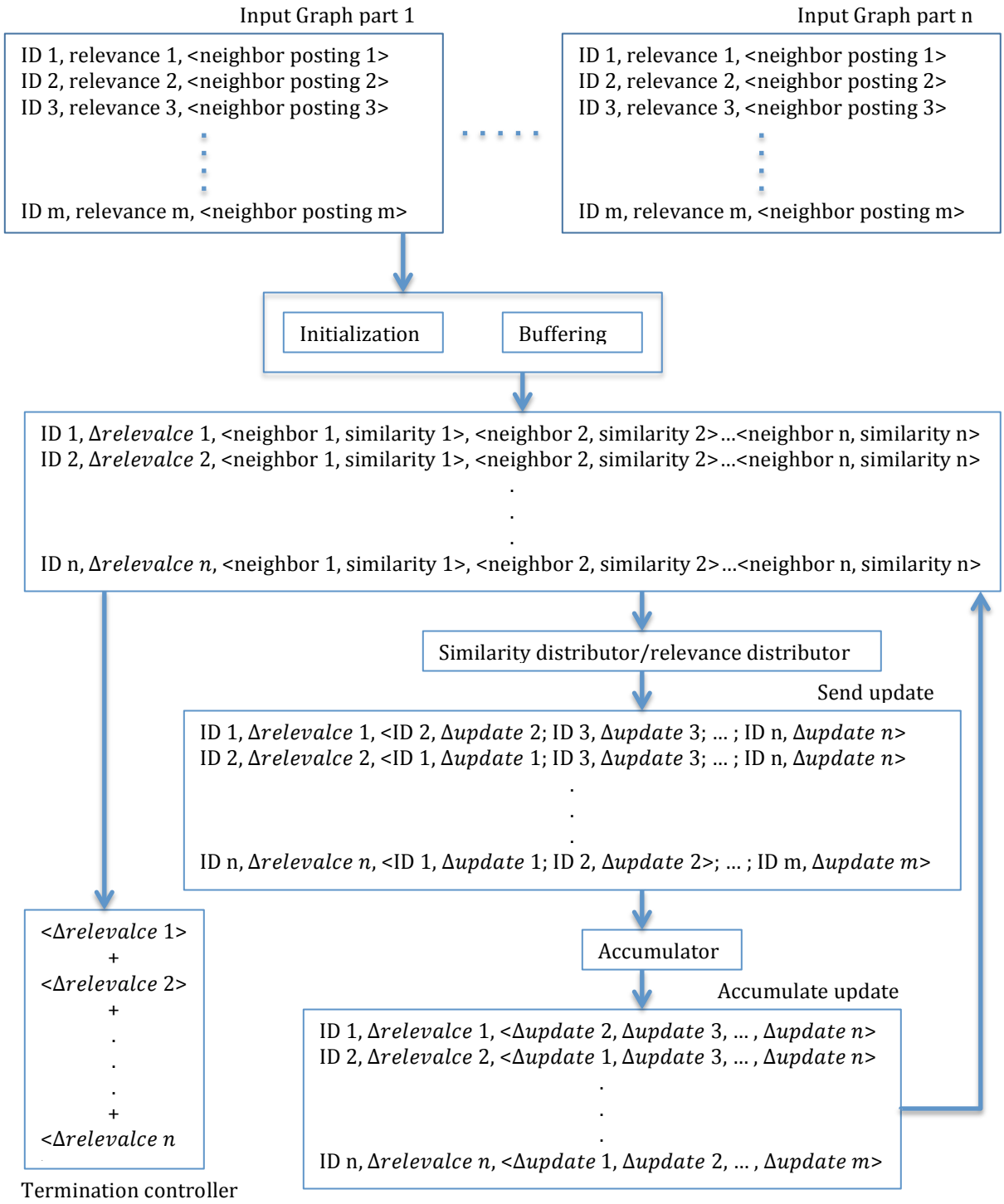


Figure 33 Maiter design

Comparing Figure 29 and Figure 31, we can easily see the difference between the MapReduce implementation and the Maiter implementation. First of all, the MapReduce design has a big loop, which is indicated by the blue arrows, and each step in the loop has to be executed one by one. However, in the Maiter design, there is no such loop. Each time the accumulator sends updates to the buffered graph, the updates will be distributed by their similarity to the accumulator. The termination check step is executed periodically and does not affect the accumulative update process at all. Second, from Figure 29, we know that the output of every iteration will be written into the HDFS, while in Maiter, the graph is buffered all the time for fast processing.

To sum up, the MapReduce model is iterative. On the contrary, the Maiter model relies on the accumulative update mechanism and does not has an iteration loop in the design. Next, the MapReduce design requires a lot of I/O process. One read and one write are needed for all iterations. In addition, communication with HDFS is slower than communication with the local disk. On the other hand, the Maiter design buffers all the intermediate results. Therefore, we expect Maiter to be faster than MapReduce.

6.3.3 Experiment

We collect 94 sample graphs, with the number of nodes ranging from 2 to 242. The x-axle in the result is the number of nodes in each sample graph. The y-axle is the running time of each graph. The red line is for the Maiter, and the blue line is for the MapReduce. Since the number of nodes in each graph is small, the variation

of the graph size does not have an apparent influence on the running time. Therefore the plot is a line that is nearly parallel to the x-axle.

The cluster has two machines (conceptually it is a cluster, even though there are just four machines), each with a quad-core CPU, 4BG's memory, and a 1TB hard drive. Figure 34 is the comparison result. The Maiter model with asynchronous accumulative update mechanism is 45 times faster then the MapReduce model with synchronized iterative mechanism in random-walk process.

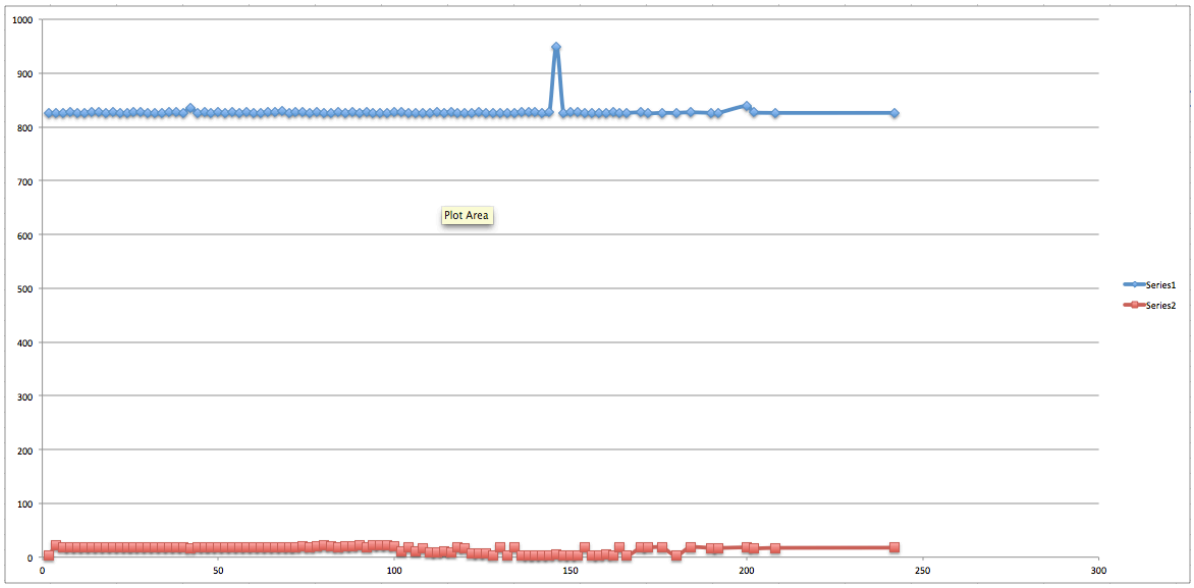


Figure 34 comparison of MapReduce and Maiter

Figure 35 is the statistic summary of the experiment. The red column is the size of each graph, ranging from 2 nodes to 94 nodes, which is incremented by 2. The first blue column is the running time of the MapReduce model, and the second column is the running time of the Maiter model. The unit is in seconds. From the experiment statistics, the Maiter takes an average of 18 seconds to process one image page, and it would estimate to take 23 hours to process 4736 pages. MapReduce is much worse since it take 826 seconds for one page. That is because

these models are designed for large-scale data, not for the tiny graph, which has only 100-200 nodes. In the next chapter, I will talk about the trick to utilize the powerful cloud environment for our 4736 tiny graphs. In this chapter, we compare MapReduce and Maiter from both the principle and experiment, and the Maiter model wins the ticket to offer computing assistant to random walk refinement.

	# node	MapReduce	Maiter
1	2	826	3.08176
2	4	826	22.1364
3	6	825	19.151
4	8	828	19.1271
5	10	825	18.1411
6	12	826	19.1277
7	14	828	18.123
8	16	827	18.1307
9	18	826	18.1236
10	20	827	18.1244
11	22	826	18.1321
12	24	826	18.1376
13	26	827	18.1321
14	28	827	18.1317
15	30	826	18.1367
16	32	825	18.1335
17	34	825	18.1374
18	36	827	18.1434
19	38	827	18.1412
20	40	825	18.1429
21	42	836	16.135
22	44	826	18.1412
23	46	827	18.1488
24	48	826	18.1495
25	50	827	19.1535
26	52	826	18.1517
27	54	827	18.1558
28	56	826	18.1587
29	58	827	18.1618
30	60	826	18.1581
31	62	826	19.1637
32	64	827	19.1696
33	66	827	19.1675
34	68	829	18.1697
35	70	825	19.1766
36	72	828	19.1828
37	74	827	20.1805
38	76	825	19.1879
39	78	827	20.1984
40	80	825	21.2025
41	82	825	20.1988
42	84	827	19.2073
43	86	826	20.2158
44	88	828	20.2144
45	90	826	21.2213
46	92	828	19.2187
47	94	826	21.2254
48	96	826	22.2524
49	98	825	21.2411
50	100	827	19.2449
51	102	827	11.1854
52	104	826	18.2532
53	106	825	10.1879
54	108	826	17.263
55	110	826	9.18703
56	112	827	8.18473
57	114	826	11.2193
58	116	828	9.20096
59	118	826	18.321
60	120	825	17.3218
61	122	826	6.17697
62	124	827	7.19703
63	126	826	7.20193
64	128	826	3.14913
65	130	826	18.4079
66	132	826	3.14817
67	134	826	18.4186
68	136	827	3.1566
69	138	827	3.16089
70	140	827	3.16911
71	142	826	3.16363
72	144	827	3.17024
73	146	949	4.19451
74	148	826	3.17637
75	150	828	3.17731
76	152	827	3.17962
77	154	826	17.5645
78	156	826	3.18744
79	158	826	3.18762
80	160	826	4.22377
81	162	827	3.19744
82	164	826	17.6628
83	166	826	3.19971
84	170	827	17.7091
85	172	825	17.7268
86	176	826	17.8062
87	180	826	3.24194
88	184	827	17.8679
89	190	826	16.905
90	192	826	16.9394
91	200	839	18.0633
92	202	827	16.0718
93	208	826	17.1449
94	242	826	17.5444

Figure 35 comparison result

CHAPTER 7

DEPLOYMENT OF RANDOM WALK PROCESS ON MAITER

7.1 Convergence

In section 4.2, we have given the expression of the random walk refinement process. The vertices are the tag relevance score, and edges are similarities between tags.

$$r_k(i) = \alpha \sum_j r_{k-1}(j) p_{ij} + (1-\alpha) v_i \quad (7.1)$$

Where

$$p_{ij} = \frac{s_{ij}}{\sum_k s_{ik}} \quad (7.2)$$

We have proved that the random walk converges in section 4.2, and derived the following equation for computation. In the experiment in chapter 5, the random walk refined is done by matrix computation using equation (7.3).

$$\mathbf{r}_\pi = (1-\alpha)(\mathbf{I} - \alpha \mathbf{P})^{-1}\mathbf{v} \quad (7.3)$$

Maiter's asynchronous accumulative update approach performs computation based on equation (7.1) without any proof on the convergence issue. The framework will run the iteration forever, and the more iterations that got run, the closer it gets to the actual convergence values. In the next paragraph, we introduce L1-norm to help evaluate the convergence state.

Assume $v = \{v_1, v_2, \dots, v_n\}$, L1-norm = $\sum_{i=1}^{n} v_i$. In the framework, we take a summation of all the values in the graph nodes as the L1-norm. After each iteration ends, we take a difference of the current L1-norm and the previous L1-norm. If the

difference is smaller than the threshold, it can be assumed close enough to the convergence state, and the framework terminates the program. The process to check convergence is called “termination check” [21].

7.2 Maiter graph generation

Maiter is designed for large-scale distributed computation. In section 6.3.3, the experiment shows that it takes 20 seconds to run a graph with 100-200 nodes with two machines. Since the communication between machines affects the performance when running on the small graph, we re-do the experiment with one single machine, and it takes 4 seconds on average to run a single image page graph. For our collection with 4736 graphs, the total execution time estimates to be 27 hours running on two machines, and 5.26 hours running on one single machine, which is even slower than the original 3.2 hours. The reason is that Maiter is designed for very large-scale computation and graph loading; as well the configuration also adds some time. It is suitable to load very large graph at one time rather than load many small graphs individually. Therefore, we combine all 4736 graphs into one big graph, with the sub-graphs unconnected to each other. From section 6.3.2, we know the way Maiter works is by continuously sending received updates, so the sub-graphs will not affect others. The only problem is that termination check may differ from running the sub-graph separately. To solve this issue, we set the termination threshold to be very small, so that the difference will be weakened significantly. Figure 36 illustrates the Maiter graph.

Since Maiter loads the entire input graph into buffer, the next concern is how to process very large input graph that cannot fit into memory. We split the graph into pieces and distribute them to different machines. There are many ways to split the intput graph, and splitting the graphs randomly is the simplest way. However, randamly splitting may also distribute the sub-graphs to different machines and therefore cause intensive machine communication. In order to minimize the communication, we group all the adjacent subgraphs, and only the last few sub-graphs in each group may corss to the machines. Figure 37 illustrates the idea.

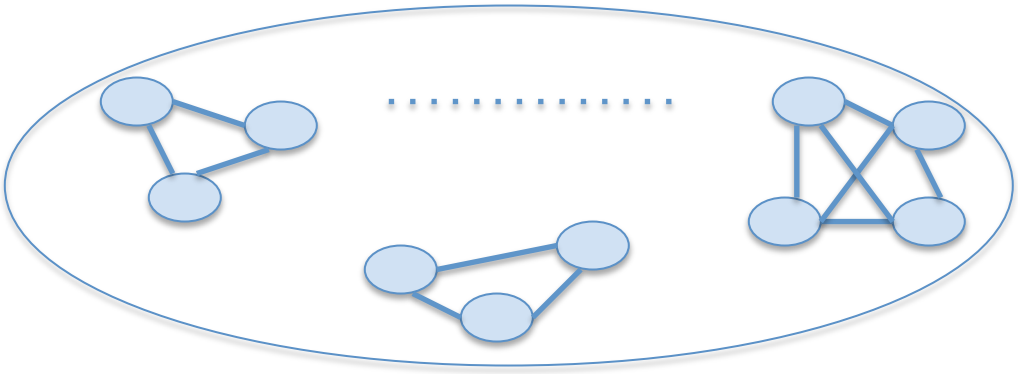


Figure 36 input graph

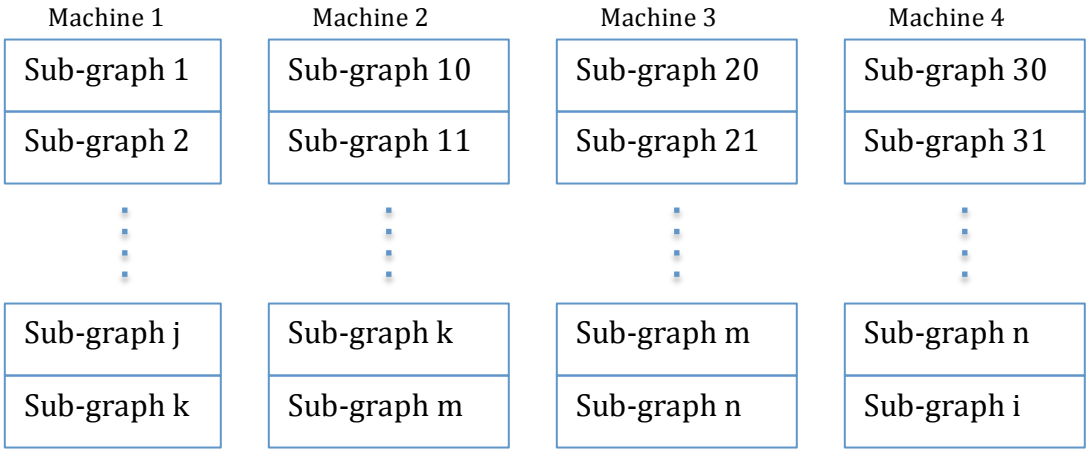


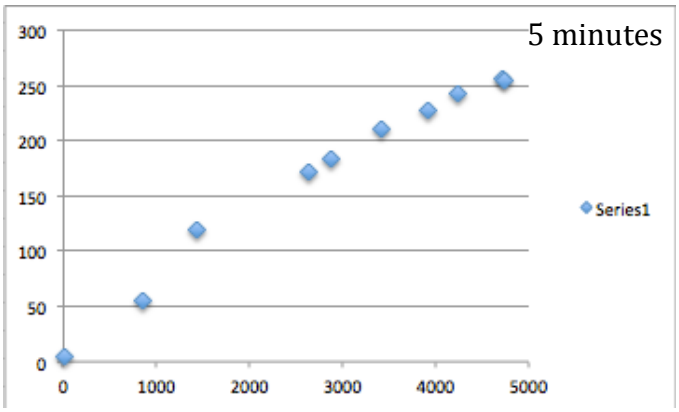
Figure 37 graph splitting

7.3 Experiment

7.3.1 Random walk

For fairness, we use one machine with a quad-core CPU, 4GB's memory and a 1TB hard drive to run the Maiter framework. I have done 10 group of experiments, with the graph size ranging from 122 nodes to 261,097 nodes, and sub-graphs number ranging from 5 to 4736. Figure 38 shows the result.

# of graphs	time
5	4
844	55.55
1440	118.892
2644	172.381
2873	183.232
3423	210.406
3921	227.766
4240	242.137
4712	255.282
4736	253.963



# of nodes	time
122	4
50046	55.55
100125	118.892
160003	172.381
170037	183.232
203011	210.406
223006	227.766
240003	242.137
260123	255.282
261097	253.963

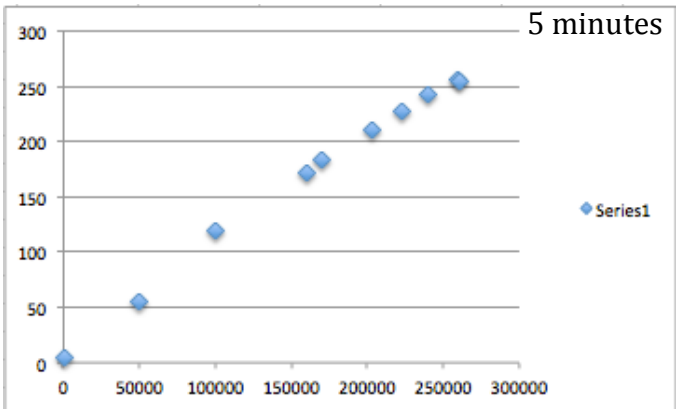


Figure 38 random walk on Maiter

Compared with the original 192 minutes, Maiter just needs 254 seconds with one single machine and 105 seconds with four machines, which is 46 times faster with one machine and 110 times faster with four machines. MapReduce is a disaster for the random walk process (7 hours) because it is not designed for iterative jobs.

7.3.2 Multi-machine performance

In this section, I compare the performance between using one single machine and using machine clusters with different graph splitting methods (random splitting and group splitting). I also do a comparison of using the whole graph input idea proposed in section 7.2 and using the sub-graph streaming input idea.

Whole graph input	One machine	Four machine
Random graph splitting	256 seconds	419 seconds
Group graph splitting	256 seconds	105 seconds

Sub-graph stream input	One machine	Four machine
Group graph splitting	18944 seconds	4736 seconds

Whole graph/sub-graph	One machine	Four machine
Improvement	74 times	45 times

Figure 39 Maiter performance comparison

CHAPTER 8

EVALUATION

8.1 Effectiveness measurement

Let me introduce two concepts first. Precision measure the percentage of relevant results found in the total number of results returned.

$$p = \frac{R(q)}{N(q)} \quad (8.1)$$

R(q) is the number of relevant results for query q. N(q) is the total number of search results for query q. For example, a list of 10 results contains 3 relevant result gives us a precision of 0.3. The length of the ranking list has a big influence on the precision calculation. Precision value before the first 10 results will always be higher than precision value after 30 results. Therefore, we hope that most relevant results could be ranked at the top.

Recall is another metric for evaluation. Recall is defined by the number of relevant pages found out of the number of relevant pages that exists. We use pages here because our page is the return unit for the image search project. It could be any kind of document.

$$r = \frac{R(q)}{RE(q)} \quad (8.2)$$

R(q) is the number of relevant pages found for query q, and RE(q) is the number of relevant pages that exists for query q.

The average precision is an attempt to combine the precision and recall. Average precision brings order to the measurement. Pages with relevant results

mostly occur on top of the ranking list, which will be assigned the highest average precision. Figure 40 give an example of how to calculate average precision. Assume we have an image of an “Apple”, and the expected relevant tags are “Apple”, “Fruit”, “Red” and “Sweet”. The length of the list begins at 1, and increment by 1 each time. We compute the precision for each sub-list, and record precision under p, as shown in figure 40. We also compute the recall at each round. Finally, we get a list like figure 28. Take the average precision for expected relevant tags, and we are done. In this example, average precision equals (1 + 0.67 + 0.6 + 0.67)/4 = 0.735.

Tags	p	r
Apple	**1**	**0.25**
Catty	0.5	0.25
Fruit	**0.67**	**0.5**
Book	0.5	0.5
Red	**0.6**	**0.75**
Sweet	**0.67**	**1**
Glass	0.57	1
Wine	0.5	1
Pencil	0.44	1
Song	0.4	1

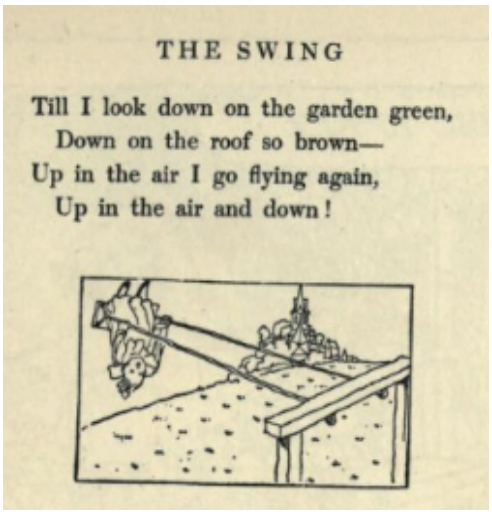
Figure 40 evaluation example

For evaluating the whole collection, we take another average of all average precision scores, and call it the “mean average precision”. We use mean average precision to evaluate the image search engine in the following experiment.

8.2 Experiment

For evaluation, we have to pick up relevant results manually. It is time consuming work. For book image search project, we have indexed 50 books, 4736 image pages. And I picked up relevant tags for 100 image pages. I did evaluation for

both initial ranking list and refined ranking list. The purpose to do both is to compare the mean average precision and thus figure out the effectiveness of the random walk refinement. Figure 41 lists some snap shot from the evaluation system interface, and you can see the ranking changes and precision changes visually.


THE SWING

Till I look down on the garden green,
Down on the roof so brown—
Up in the air I go flying again,
Up in the air and down!

average precision before random walk	average precision after random walk
0.22619047619047616	0.41666666666666663
air	air
brown	roof
green	**swing**
garden	**flying**
roof	garden
swing	green
flying	brown

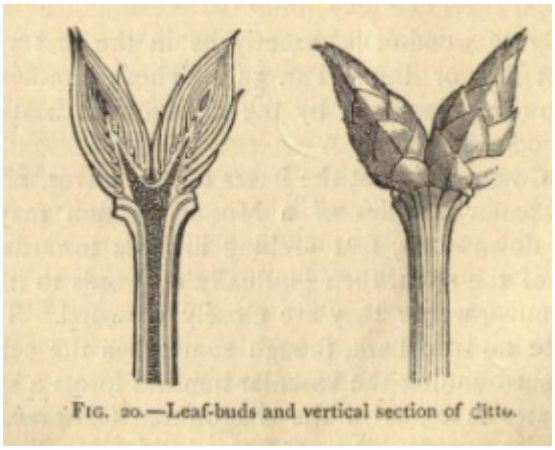


FIG. 20.—Leaf-buds and vertical section of Ditto.

before random walk	after random walk
0.4488562091503268	0.6099567099567099
buds	**buds**
branches	branches
terminal	**stem**
stem	terminal
wood	spring
formed	trees
nds	**plants**
plants	**stems**
described	bark
angle	autumn
covered	**leaf**
complete	remain
vertical	dormant
spring	ends
arrangement	wet
cold	covered
trees	union
leaf	characteristic

Figure 41 evaluation result example

8.3 Evaluation summary

Statistics summary for effectiveness

Total pages evaluated: 135

Average precision increased after random walk refinement: 85 pages

Average precision decreased after random walk refinement: 50 pages

Mean average precision before random walk refinement: 0.28

Mean average precision after random walk refinement: 0.304

Mean average precision improvement after random walk refinement: 8.6%

Statistics summary for efficiency

Total number of pages tested: 4736

Total graph size 261,097 nodes

Random walk execution time: 192 minutes

Random walk execution time with Maiter: 4.2 minutes

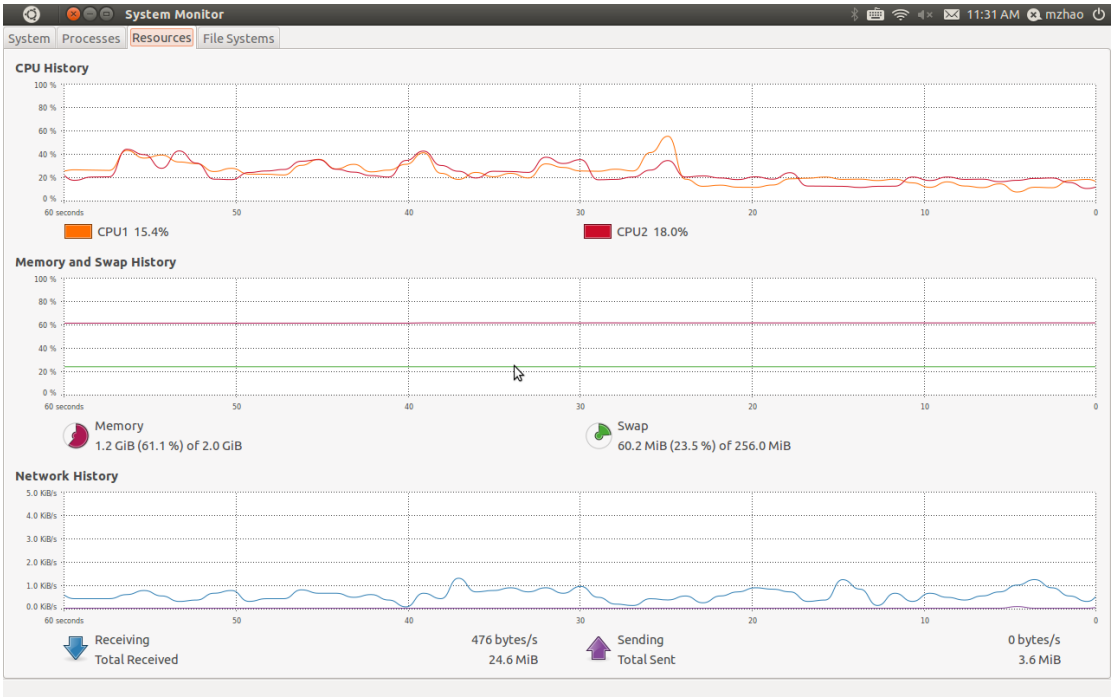
Random walk execution time improvement: 46 times faster

APPENDIX A

EFFICIENCY ANALYSIS

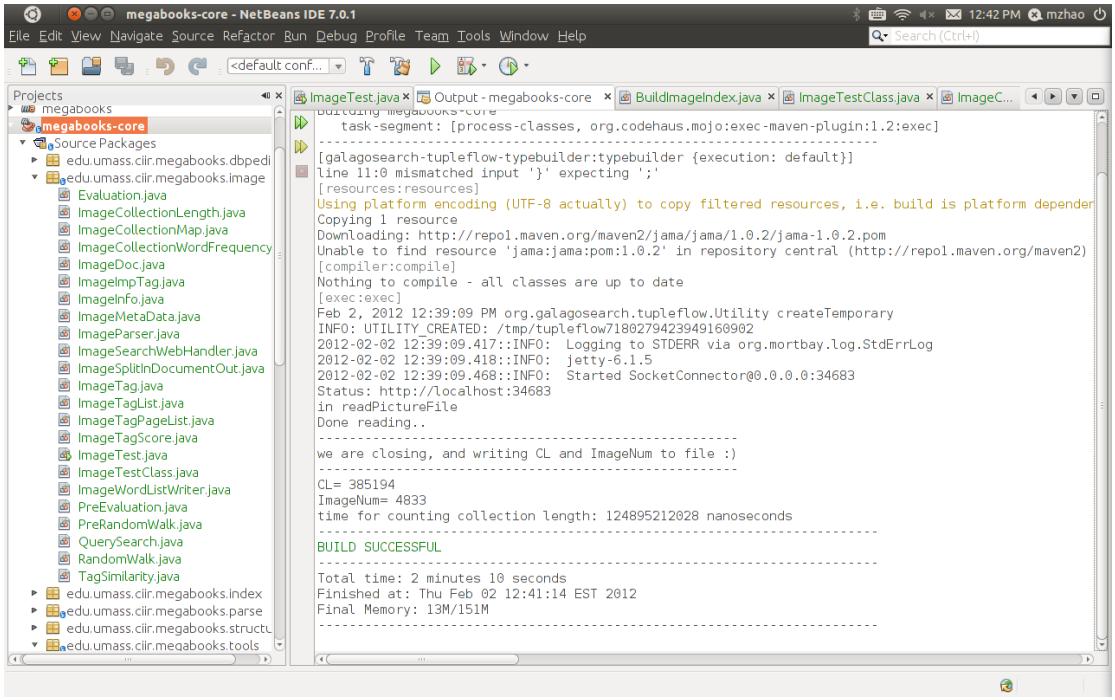
STEP 1: Counting collection length

Before execution of STEP 1



Memory usage: 1.2 out of 2.0 GB, 61.1% occupancy

During and after execution of STEP 1

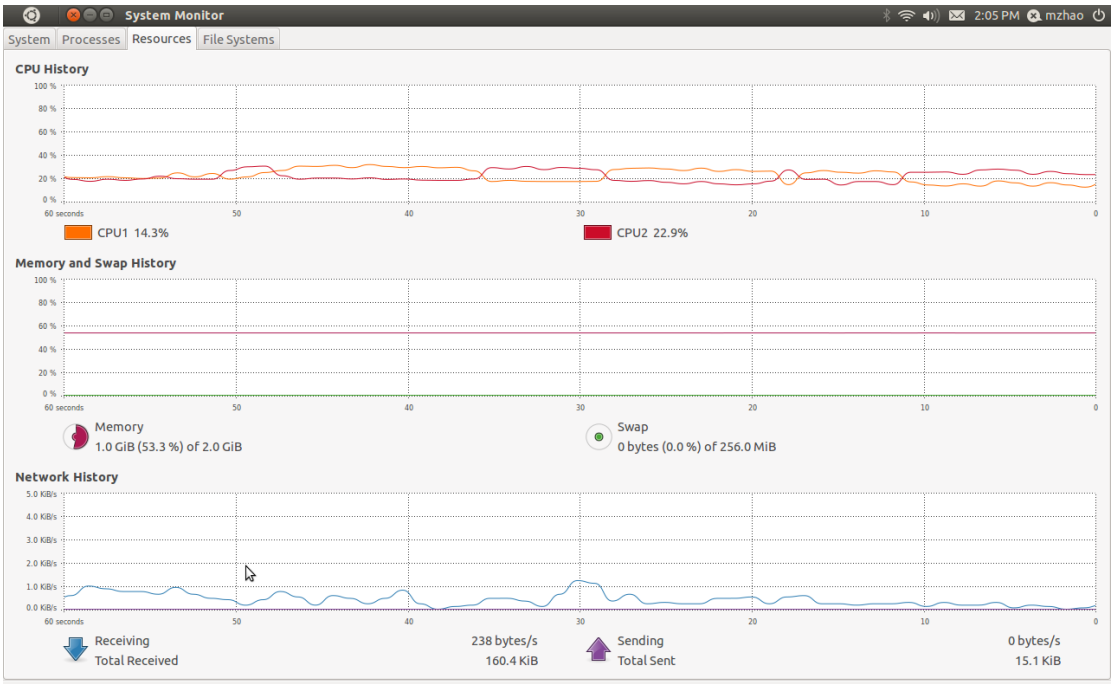


Memory usage: 1.8 out of 2.0 GB, 92.3% occupancy

Total running time: 125 seconds

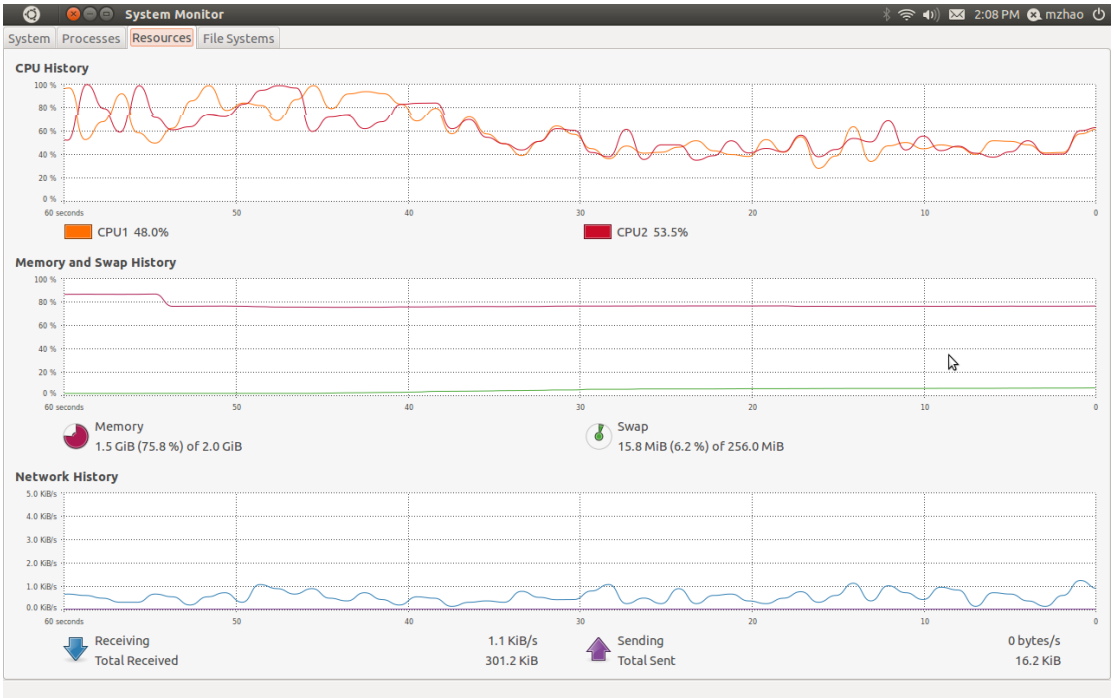
STEP 2: Collecting identifier list for each tag

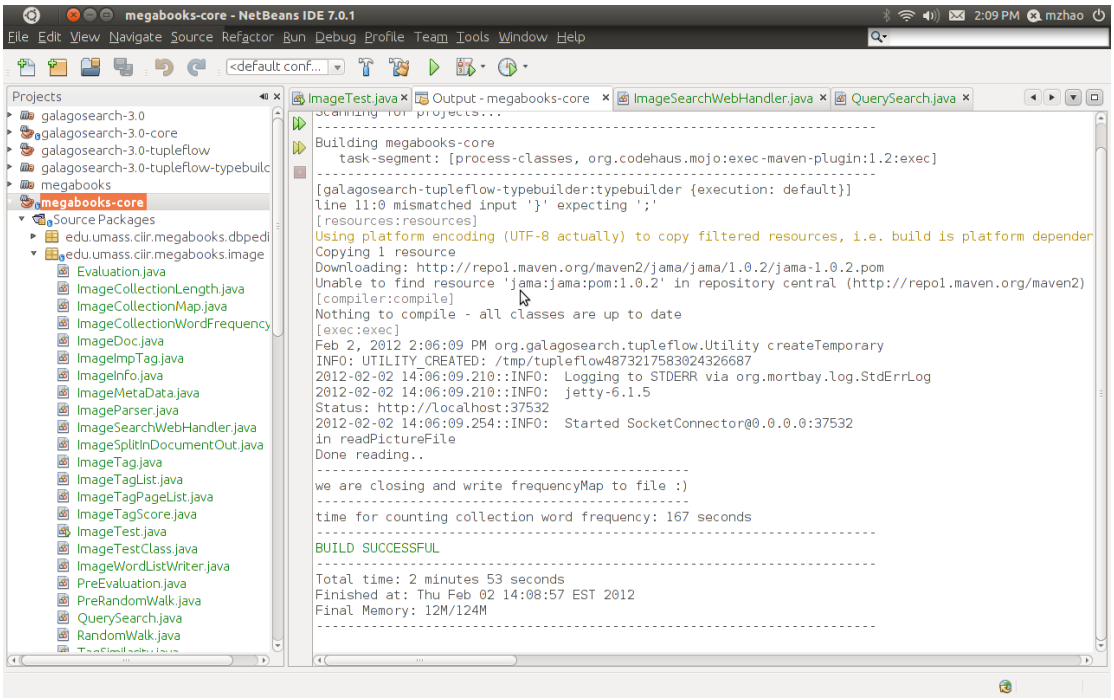
Before execution of STEP 2



Memory usage: 1.0 out of 2.0 GB, 53.3% occupancy

During and after execution of STEP 2



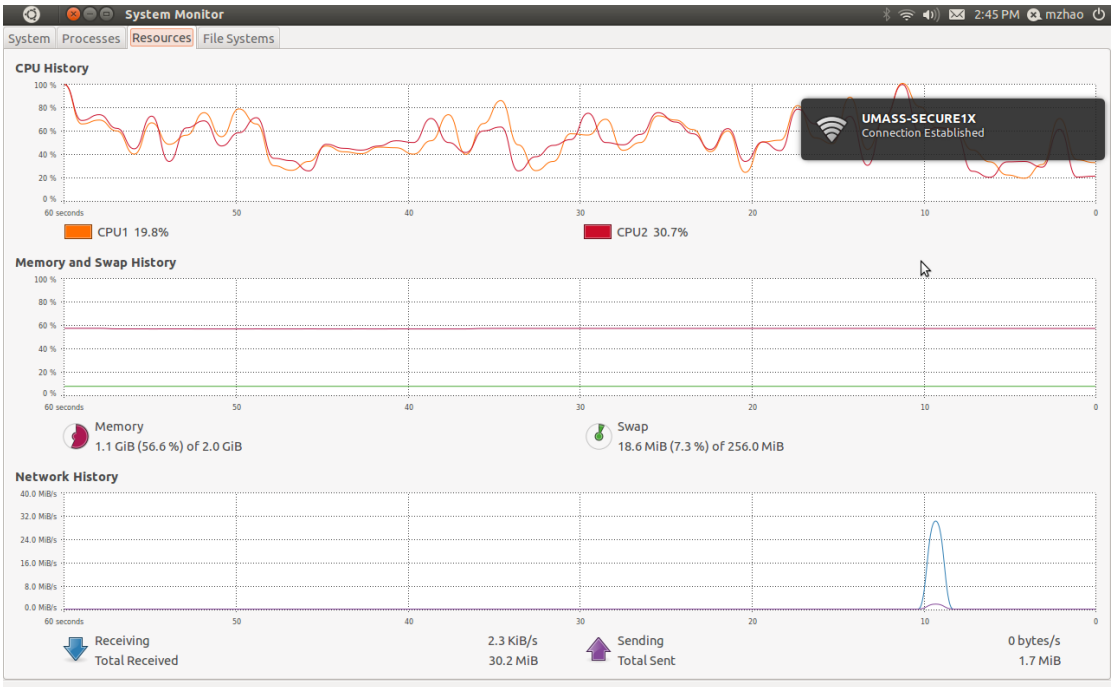


Memory usage: 1.5 out of 2.0 GB, 75.8% occupancy

Total running time: 167 seconds

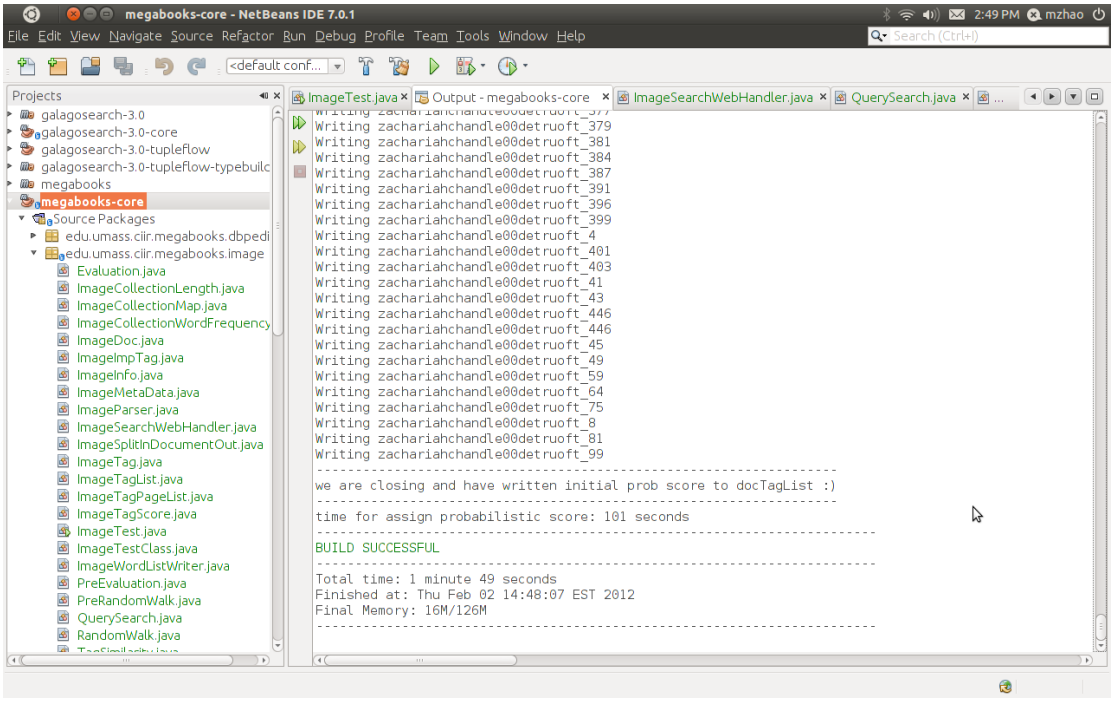
STEP 3: Apply probabilistic model and create ranked tag list

Before execution of STEP 3



Memory usage: 1.1 out of 2.0 GB, 56.6% occupancy

During and after execution of STEP 3

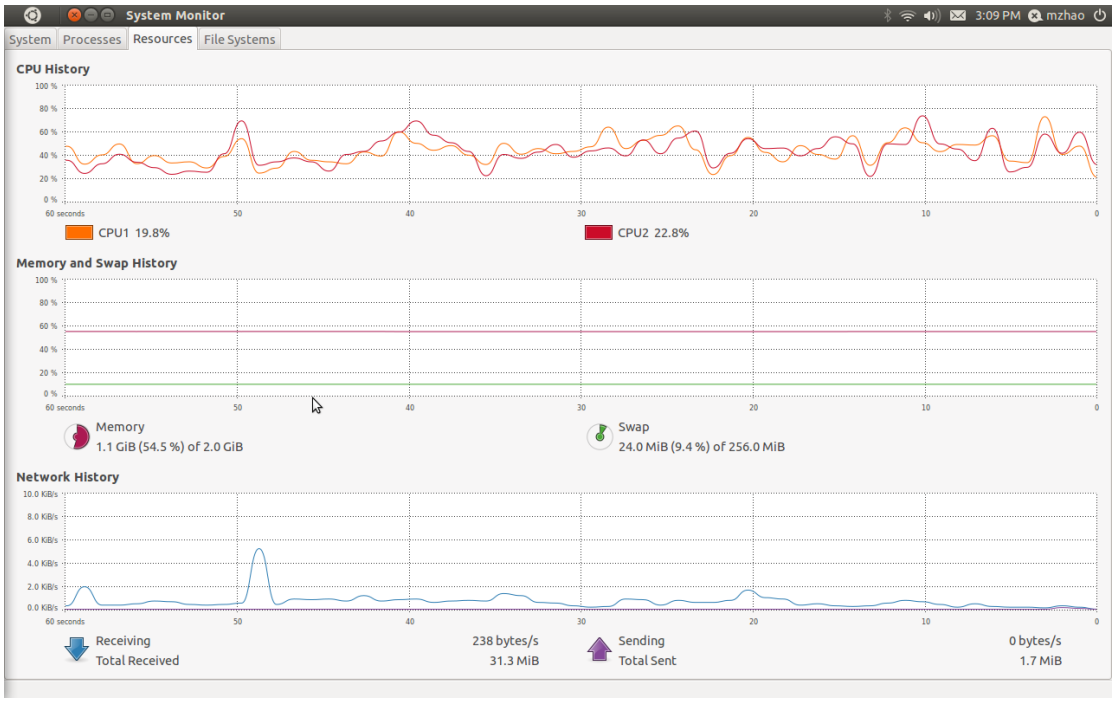


Memory usage: 1.6 out of 2.0 GB, 79.6% occupancy

Total running time: 101 seconds

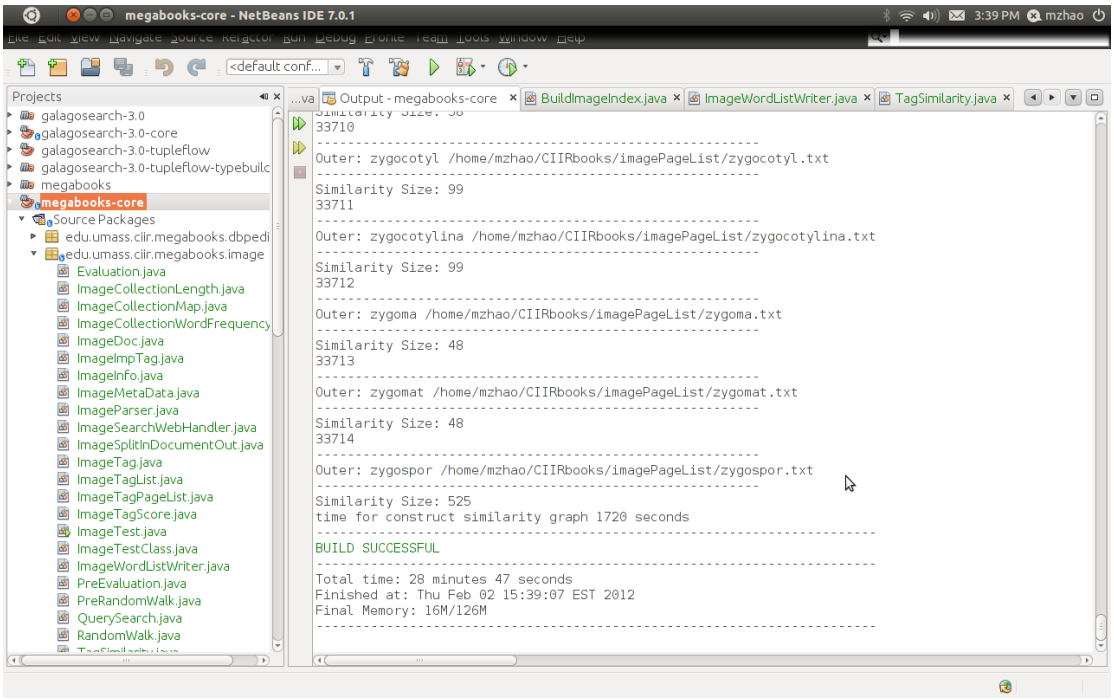
STEP 4: Similarity graph construction

Before execution of STEP 4



Memory usage: 1.1 out of 2.0 GB, 54.5% occupancy

During and after execution of STEP 4

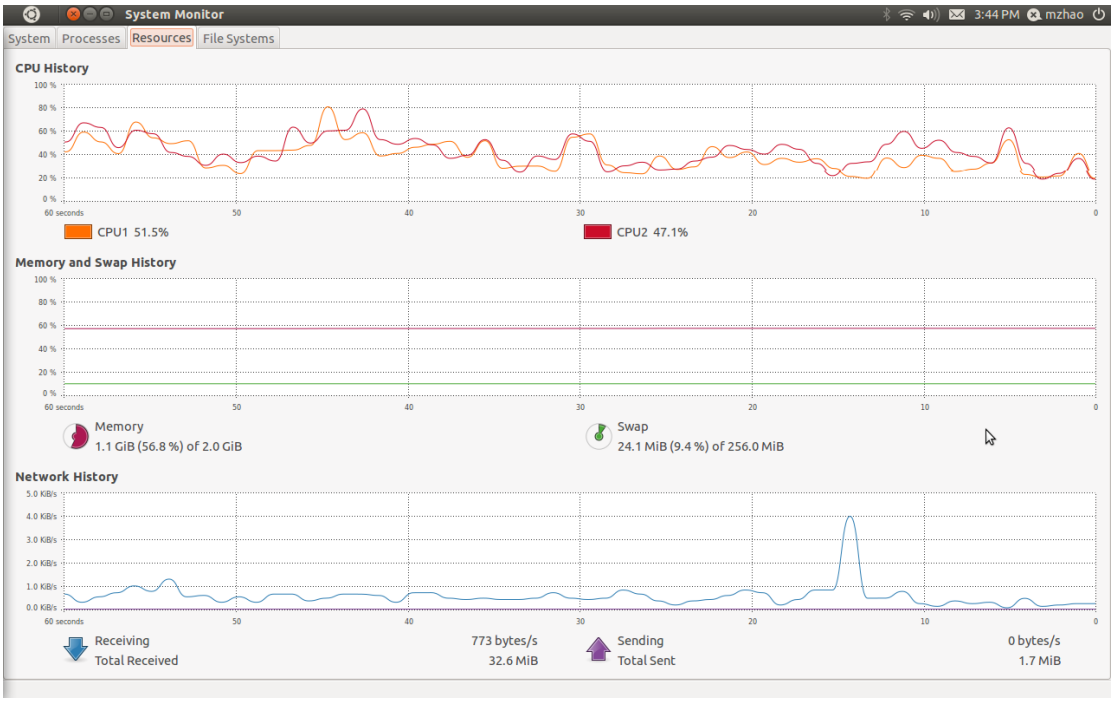


Memory usage: 1.3 out of 2.0 GB, 64.7% occupancy

Total running time: 1720 seconds (29 minutes)

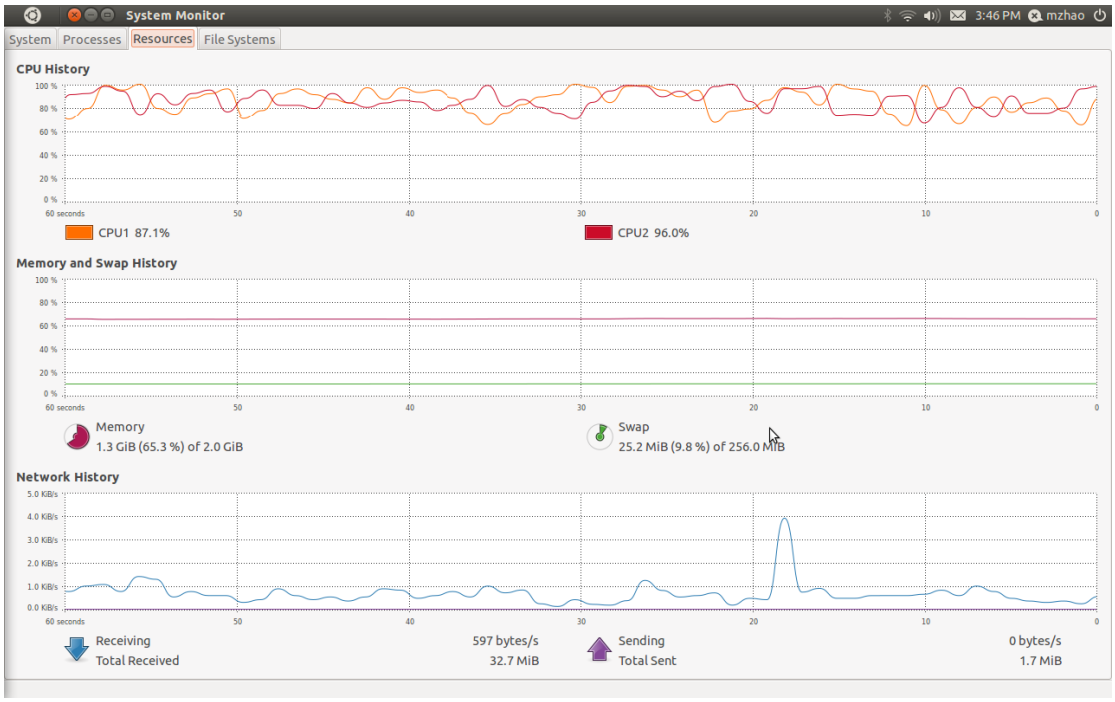
STEP 5: Random walk refinement

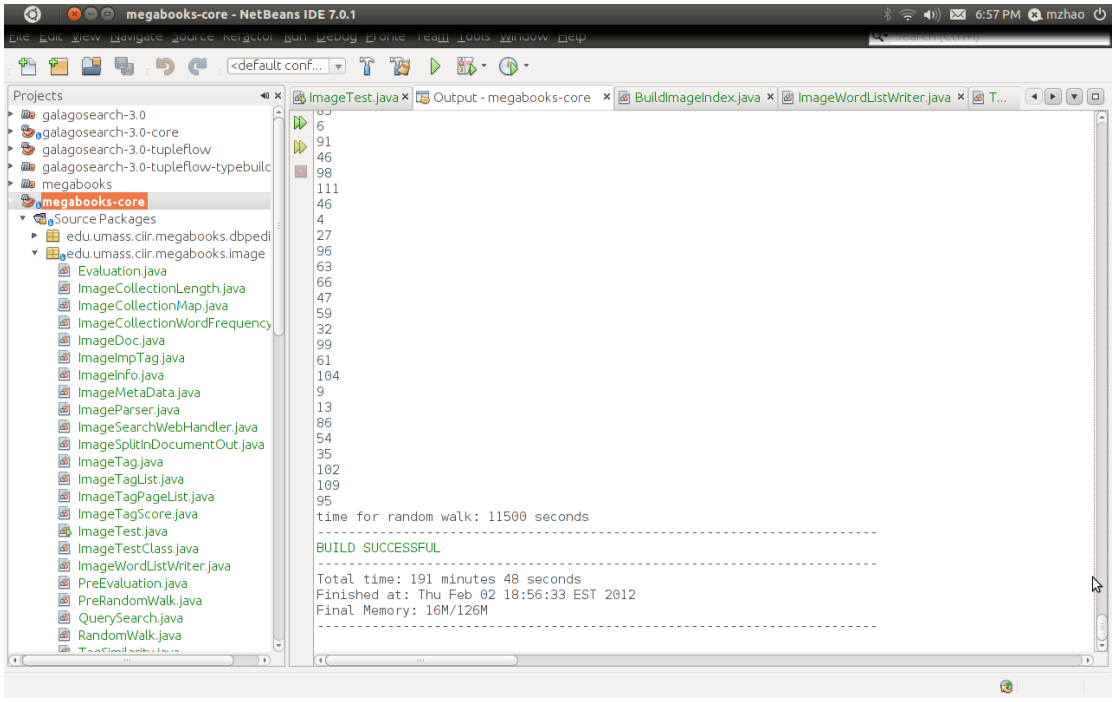
Before execution of STEP 5



Memory usage: 1.1 out of 2.0 GB, 56.8% occupancy

During and after execution of STEP 5





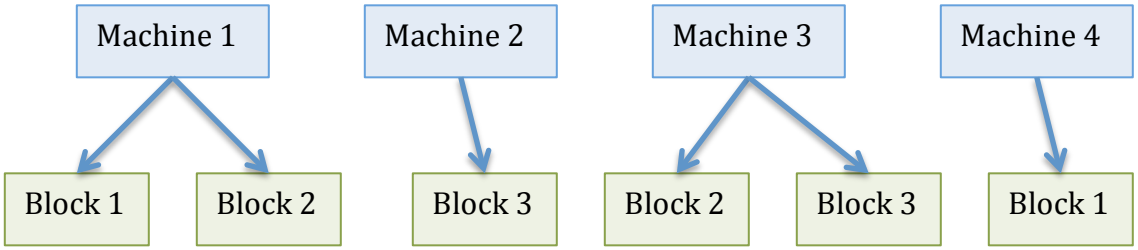
Memory usage: 1.3 out of 2.0 GB, 65.3% occupancy

Total running time: 11500 seconds (192 minutes)

APPENDIX B

HADOOP FRAMEWORK

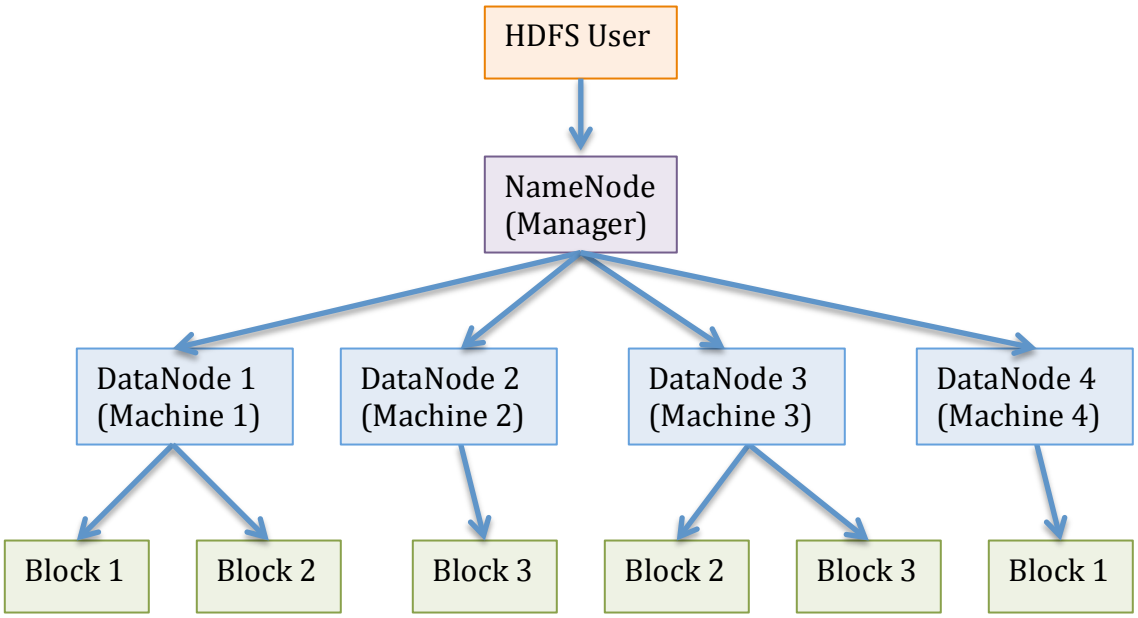
Before I dive into the architecture of the implementation, I would like to talk a little about the Hadoop Distributed File System (HDFS) that associates with the framework. As we know, the data centers have hundreds or thousands of machines, and these machines are always inexpensive. Therefore, hardware failures should be very common and HDFS is responsible for failure detection and recovery. HDFS uses replication to provide reliable storage. Each file has been chopped into small blocks with the same size, which is usually 64 MB, except the last one [18]. The blocks are replicated in case of hardware failure.



HDFS storage

HDFS consists of a manager, called NameNode, and a cluster of workers, called DataNode [18]. NameNode is responsible for the access control and general file system management. DataNode, as implied by the name, is mainly for data storage, data read, write, deletion and replication under the order of NameNode. View HDFS from the high level, it is nearly the same as operating on the single machine. Though when view from the lower level, it differs in the storage mechanism. HDFS does allow user data to be stored in files, but files are chopped into blocks, and stored in different machines. This design provides file system

reliability and fault tolerant ability. When performing operations on the file system, user communicates with the NameNode, and the NameNode sends orders to the DataNode to complete the user command since the NameNode knows exactly the entire mappings among files, blocks and machines in the low level.



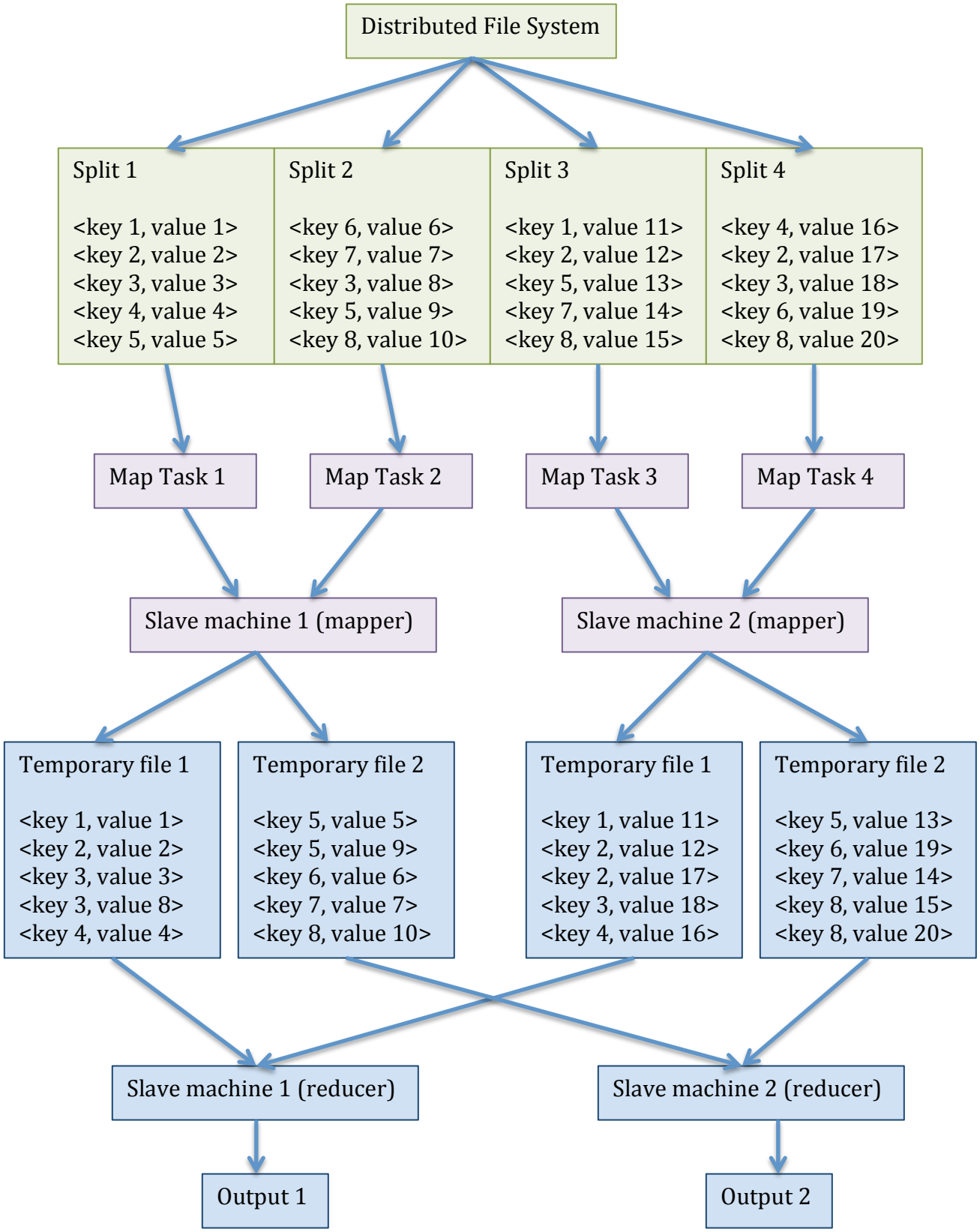
HDFS structure

I mentioned in the previous paragraph that HDFS looks the same as the normal desktop from the high level, because HDFS is also using hierarchical file organization. All the block operations are hided and users can perform copy, deletion, move operations and create directories. The NameNode is in charge of holding and maintaining the file system namespace, and it records every change made on namespace. File system namespace will be loaded into memory, and NameNode checks namespace table frequently for mapping among metadata, files and block locations [18, 19].

After I have covered the Hadoop distributed file system, we can go back to the Hadoop framework implementation now. In my MapReduce experiment, there

are four machines with a quad-core processor, 4GB's memory, and 1TB disk running Linux. There is no hardware failure in my experiment, so I am not talking more about fault tolerance and recovery any more. Figure 33 is the high level architecture of Hadoop frame, and the following paragraph describes how it works.

First, the framework is deployed on a cluster of machines, therefore, we need a manager to coordinate their work, and this manager is called "Master". The rest of the machines are called "worker" or "slave". Second, the framework takes the input file as a job, and chops the whole input into small pieces, called "splits". Each split is usually 64MB (one block in HDFS is 64 MB), and each split will be processed as a map task. The master assigns map tasks to each machine. When the slave machine receives the order, it retrieves a corresponding input split and creates a map task. In the map task, key value pairs are parsed and passed to the map function. Then the intermediate key value pairs are produced, partitioned and written into temporary local files. Values associated with the same key have to be partitioned into the same reducer (slave machines that perform reduce tasks). This is the rule; otherwise, key list pair cannot be generated properly. Simple rules such as MOD work for the partition. Third, after the map tasks finish, the reducers read the temporary files and sort the intermediate key value to group values with the same key. Finally, reducers apply reduce function to each key list pair and generate output. In the whole process, the master is responsible for distributing tasks (both map tasks and reduce tasks), storing task states, and temporary file locations, etc. The light green part is the input splits, the light purple part is the map process, and blue part is the reduce process.



MapReduce flow chart

BIBLIOGRAPHY

[1] Hao Xu, Jingdong Wang, and Xiansheng Hua. *Interactive image search by 2D semantic map. In WWW 's 10: Proceedings of the 19th international conference on World Wide Web.* New York, NY: ACM Press, 2010.

[2] Ying Liu, Tao Qin, Tieyan Liu, Lei Zhang, and Weiying Ma. *Similarity space projection for web image search and annotation. In MIR '05: Proceedings of the 7th ACM SIGMM international workshop on Multimedia information retrieval.* New York, NY: ACM Press, 2005.

[3] Dong Liu, Xiansheng Hua, Linjun Yang, Meng Wang, and Hongjiang Zhang. *Tag ranking. In WWW '09: Proceedings of the 18th international conference on World Wide Web.* New York, NY: ACM Press, 2009.

[4] Jingyu Cui, Fang Wen, and Xiaoou Tang. *Real time google and live image search re-ranking. In MM '08: Proceedings of the 16th ACM international conference on Multimedia.* New York, NY: ACM Press, 2008.

[5] Kambiz and Ling Guan. *Content-based image retrieval via distributed databases. In CIVR '08: Proceedings of the 2008 international conference on the content-based image and vedio retrieval.* New York, NY: ACM Press, 2008.

[6] Trevor Strohman. *Efficient processing of complex feature for information retrieval. PhD dissertation, University of Massachusetts.* Amherst, MA, 2007.

[7] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, Matei Zaharia. *Above the Clouds: A Berkeley View of Cloud Computing.* Berkeley, CA, 2009.

[8] *Cloud Computering [online]. Available from: http://en.wikipedia.org/wiki/Cloud_computing.*

[9] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters. In OSDI 's04: 6th Symposium on Operating Systems Design and Implementation.*

[10] *MapReduce [online]. Available from: http://en.wikipedia.org/wiki/MapReduce.*

[11] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schutze. *Introduction to Information Retrieval.* Cambridge University Press, 2008.

[12] Bruce Croft, Don Metzler, and Trevor Strohman. *Search engine, information retrieval in practice.* Addison Wesley, 2010

[13] Donald Metzler and Bruce Croft. *A markov random field model for term dependencies. In SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on research and development in information retrieval.* New York, NY: ACM Press, 2005.

[14] *Internet [online]. Available from: http://en.wikipedia.org/wiki/Internet.*

[15] *Facebook [online]. Available from: http://en.wikipedia.org/wiki/Facebook.*

[16] *LinkedIn [online]. Available from: http://en.wikipedia.org/wiki/Linkedin.*

[17] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, Matei Zaharia. *A view of cloud computing.* Vol. 53. New York, NY, 2010.

[18] *HDFS architecture guide [online]. Available from: http://hadoop.apache.org/common/docs/r1.0.3/hdfs_design.html.*

[19] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. *The google file system. In SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles.* New York, NY, 2003.

[20] Sergey Brin and Lawrence Page. *The anatomy of a large-scale hypertextual web search engine.* In WWW7 Proceedings of the seventh international conference on World Wide Web 7, Page 107-117, 1998

[21] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Accelerate large-scale iterative computation through asychronous accumulative updates. In Proceedings of the 3rd workshop on Scientific Cloud Computing in conjuction with HPDC 2012, Delft, Netherlands, June 2012