Report

Zhangjie Cao
School of Software
Tsinghua University
Beijing, China
+8618401653023, 100084
caozhangjie14@gmail.com

Danrui Qi
School of Software
Tsinghua University
Beijing, China
+8613121810735, 100084
qidanrui@gmail.com

1. SYSTEM OVERVIEW

1.1 Conceptual Overview

Our system framework is designed as Figure 1 shows. And we will introduce our system implementation in terms of extracting data, analyzing data, indexing, sorting, querying and clustering

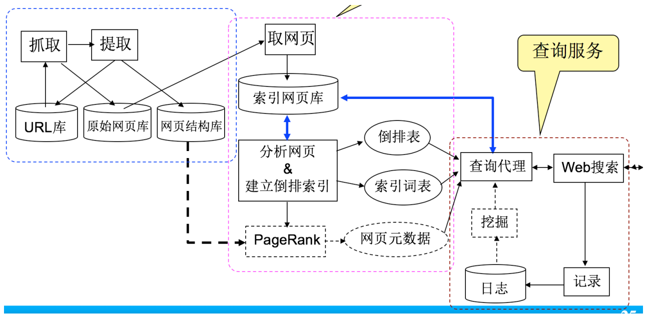


Figure1. System Framework of a Web Search Engine

1.2 Implementation Environment

- Programming Language: Python2.7
- Web Framework: Flask
- Backend Framework: Spark 2.0.1 + Hadoop 2.7
- Servers: 3 64G Memory 8 Cores Aliyun Cloud Server

2. IMPLEMENTATION DETAILS

2.1 Extracting Data

For extracting data, we use Scrapy to crawl webpages. We crawl pages from multiple anchor(root url) and extend to other pages through the href of a flag. And we only extract content of a page as one document and filter all the non-ascii characters.

2.2 Analyzing Data

After extracting data, we split text content of webpages into single words. Then we use these words to build a dictionary. For reducing time complexity of following algorithm, we only use words which occurs over 5 times

2.3 Indexing

2.3.1 Inverted Index

In computer science, an inverted index (also referred to as postings file or inverted file) is an index data structure storing a mapping from content, such as words or numbers, to its locations in a database file, or in a document or a set of documents (named in contrast to a Forward Index, which maps from documents to content). The purpose of an inverted index is to allow fast full text searches, at a cost of increased processing when a document is added to the database.

The inverted index data structure is a central component of a typical search engine indexing algorithm. A goal of a search engine implementation is to optimize the speed of the query: find the documents where word X occurs. Once a forward index is developed, which stores lists of words per document, it is next inverted to develop an inverted index. Querying the forward index would require sequential iteration through each document and to each word to verify a matching document. The time, memory, and processing resources to perform such a query are not always technically realistic. Instead of listing the words per document in the forward index, the inverted index data structure is developed which lists the documents per word.

2.3.2 Build Index

We use inverted index to index webpages. We use the words in the word dictionary in the previous step. Every word is the key of the index and the value of each key is a list of all the documents containing this word. We record their appearing times and appear places of each words in the document as the feature of document in the index.

2.4 Sorting

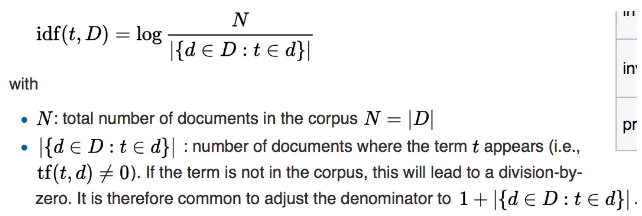
2.4.1 TF-IDF

2.4.1.1 Term Frequency

Suppose we have a set of English text documents and wish to determine which document is most relevant to the query "the brown cow". A simple way to start out is by eliminating documents that do not contain all three words "the", "brown", and "cow", but this still leaves many documents. To further distinguish them, we might count the number of times each term occurs in each document and sum them all together; the number of times a term occurs in a document is called its *term frequency*.

2.4.1.2 Inverse Document Frequency

As mentioned example above, because the term "the" is so common, term frequency will tend to incorrectly emphasize documents which happen to use the word "the" more frequently, without giving enough weight to the more meaningful terms "brown" and "cow". The term "the" is not a good keyword to distinguish relevant and non-relevant documents and terms, unlike the less common words "brown" and "cow". Hence an *inverse document frequency* factor is incorporated which diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely. IDF is calculated as following:



$$\mathrm{idf}(t,D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

with

- N: total number of documents in the corpus $N = |D|$
- $|\{d \in D : t \in d\}|$: number of documents where the term t appears (i.e., $\mathrm{tf}(t,d) \neq 0$). If the term is not in the corpus, this will lead to a division-by-zero. It is therefore common to adjust the denominator to $1 + |\{d \in D : t \in d\}|$.

in

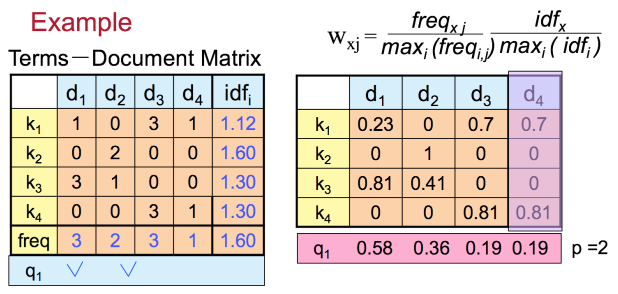
pr

2.4.1.3 TF-IDF

The TF-IDF is calculated as:

$$\mathrm{tfidf}(t,d,D) = \mathrm{tf}(t,d) \cdot \mathrm{idf}(t,D)$$

2.4.2 Weight Matrix



Example

Terms—Document Matrix

$$w_{xj} = \frac{freq_{xj}}{max_i\,(freq_{i,j})} \frac{idf_x}{max_i\,(idf_i)}$$

	d_1	d_2	d_3	d_4	idf_i
k_1	1	0	3	1	1.12
k_2	0	2	0	0	1.60
k_3	3	1	0	0	1.30
k_4	0	0	3	1	1.30
freq	3	2	3	1	1.60
q_1	∨	∨			

	d_1	d_2	d_3	d_4
k_1	0.23	0	0.7	0.7
k_2	0	1	0	0
k_3	0.81	0.41	0	0
k_4	0	0	0.81	0.81
q_1	0.58	0.36	0.19	0.19

p =2

Here we give an example to illustrate how to get weight matrix. We can get term-document matrix easily. Then, using formula above, we can get terms-document weight matrix.

2.4.3 Similarity Vector

From example in Section 2.4.2, we can see that using Euclidean distance, it is easy to get similarity between query and each document. These similarity values can be combined into a similarity vector.

2.4.4 Sorting Webpages Combining with Inverted Index

After getting similarity vector, we sort webpages combining with inverted index. First, we sort similarity of documents in inverted indexing result. Then we return them as first branch result. Secondly, we sort similarity of other documents to get the second brand result.

2.5 Clustering

2.5.1 Clustering algorithm

We use Kmeans clustering algorithm. We first extract TF-IDF features from documents as the feature of clustering. Then we set the clusters num as 100(roughly 1/1000 of the number of the documents). We initialize the centers with mass center methods as referred in the course. Then we cluster the documents.

2.5.2 Cluster label generation

We set the most 10 significant bit of center (which is a vector with length as the length of word dictionary) and set the words corresponding to these bits as the label of the cluster of this center.

3. Main Focus of System

3.1 Sparse Vector

As each document only contains a small part of the words of the word dictionary. Thus the TF-IDF feature of each document of sparse. Based on this observation, we transfer all the calculation of dense vector to sparse vector to speed up calculation.

3.2 Spark Cache

Some data structures, such as inverted index and TF-IDF feature matrix are read frequently. And their sizes are relatively stable. Thus we cache these structures in the memory (using spark cache) to speed up I/O. And at the same time, we don't cache the document content since they are only queried when the corresponding document is selected to return to the user. With these strategies, we use the memory efficiently and speed up the query.

3.3 Spark Lazy Evaluation

Spark has lazy evaluation feature. Once we declare to calculate something, spark doesn't calculate this immediately. Only when we requiring the value of the calculation will it calculate the result needed. With this feature, we divide all the returned results in multiple pages as most search engines do. Once the front end needs the result of a query, we return the first 20 results as the first page. So the back end only need calculate the 20 most similar documents first. And when the user query the other documents as he transfer to another page. We then calculate another page. This can reduce the time of the returning of the first page and avoid the user waiting too much time when he only needs the first page of the first few page, which is what most users want.