Spring 2-5-2017

Intelligent Web Crawler for Semantic Search Engine

Shujia Zhang
San Jose State University

Intelligent Web Crawler for Semantic Search Engine

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment of the Requirements for the Degree

Master of Computer Science

By

Shujia Zhang

Fall 2016

The Designated Committee Approves the Writing Project Titled

Intelligent Web Crawler For Semantic Search Engine

By Shujia Zhang

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE SAN JOSÉ

STATE UNIVERSITY

Nov 2016

Dr. Tsau Young Lin Department of Computer Science

Dr. Chris Pollett Department of Computer Science

Dr. Philip Heller Department of Computer Science

Abstract

A Semantic Search Engine (SSE) is a program that produces semantic-oriented concepts from the Internet. A web crawler is the front end of our SSE; its primary goal is to supply important and necessary information to the data analysis component of SSE. The main function of the analysis component is to produce the concepts (moderately frequent finite sequences of keywords) from the input; it uses some variants of TF-IDF as a primary tool to remove stop words. However, it is a very expensive way to filter out stop words using the idea of TF-IDF. The goal of this project is to improve the efficiency of the SSE by avoiding feeding junk data (stop words) to the SSE. In this project, we classify formally three classes of stop words: English-grammar-based stop words, Metadata stop words, and Topic-specific stop words. To remove English-grammar-based stop words, we simply use a list of stop words that can be found on the Internet. For Metadata stop words, we create a simple web crawler and add a modified HTML parser to it. The HTML parser is used to identify and remove Metadata stop words. So, our web crawler can remove most of the Metadata stop words and reduce the processing time of SSE. However, we do not know much about Topic-specific stop words. So, Topic-specific stop words are identified by a randomly selected sample of documents, instead of identifying all keywords (equal or above a threshold) and all stop words (below the threshold) on the whole set of documents. MapReduce is applied to reduce the complexity and find Topic-specific stop words such as “acm” (Association for Computing Machinery) that we find on IEEE data mining papers. Then, we create a Topic-specific stop word list and use it to reduce the processing time of SSE.

ACKNOWLEDGEMENTS

I would like to thank my academic advisor Dr. Tsau Young Lin for his meticulous guidance and endless support.

I would like to thank Dr. Chris Pollett and Dr. Philip Heller for serving on my Master's committee and helping me to stay on the right track.

I would like to thank my family for their endless encouragement and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. **INTRODUCTION**

A Semantic Search Engine (SSE) is a program that produces semantic-oriented concepts from the Internet. Web crawler is one of the main components for our SSE. The main functionality of a basic web crawler is to retrieve the HTML pages for SSE. However, the main problem is that all those data from HTML pages may contain a lot of unnecessary words that we call stop words. Stop words will slow down the SSE and affect the result produced by the SSE. A standard search engine (SE) divides keywords and stop words using Term Frequency-Inverse Document Frequency (TF-IDF) and then indexes keywords. Our SSE finds text patterns, which are called concepts, in documents using methods imported from data mining and then indexes concepts. These concepts almost correspond to frequent itemsets. Namely, concepts are very close to frequent itemsets minus generalized stop words, which are high frequent concepts. To remove generalized stop words and find the relevant concepts, the analysis component of our SSE uses some variant of TF-IDF to determine how valuable a concept is to a document in a collection of documents. However, computing TF-IDF to remove generalized stop words slows SSE significantly. The standard SE focuses on finding and indexing keywords (frequent 1-Itemsets), and the SSE focuses on producing concepts (frequent n-Itemsets).

The primary goal of this project is to reduce the cost of data analysis and improve our SSE's efficiency by discarding all types of generalized stop words. There are three classes of stop words: English-grammar-based stop words, Metadata stop words, and Topic-specific stop words. Since any existing English stop word list can remove English-

grammar-based stop words, we will focus on removing Metadata stop words and Topic-specific stop words. We want to create an intelligent web crawler which can remove unnecessary Metadata stop words from crawling procedure and find key concepts before passing the input files into the search engine. Our approach is to add an HTML parser and TF-IDF tool into the web crawler. The HTML parser can parse HTML files and remove the Metadata stop words before storing the HTML files as text files into local storage. For removing Topic-specific stop words, we will do an experiment using 10000 documents related to data mining and use MapReduce to understand and find Topic-specific stop words in the field of data mining and create a domain Stop Word List to filter them instead of computing TF-IDF to discard them.

This project is organized as follows: Section 2 describes Term Frequency-Inverse Document Frequency. Section 3 introduces stop words and describes three classes of stop words. Section 4 describes how our web crawler removes Metadata stop words. Section 5 describes Topic-specific stop words, how we create a domain stop word list, the result of an experiment and analysis. The conclusion is outlined in Section 6.

2. TERM FREQUENCY-INVERSE DOCUMENT FREQUENCY

Term Frequency-Inverse Document Frequency (TF-IDF) is a very important concept in information retrieval (IR) [5] and text mining. TF-IDF is a value that represents how important a token is to a document in a collection of documents. A token

can be a primary keyword if it appears many times in a document, but it can also be a stop word or a common word that repeats many times in a collection of documents.

2.1 Term Frequency (TF)

Term Frequency is a frequency value that indicates how frequently a token appears in a document. Since every document is very likely different in size, a token can appear much more times in longer documents than shorter ones. Therefore, we use the number of times a token occurs in a document divided by the total number of tokens in a document to compute a token's TF value.

For instance, "data mining" appears 50 times in a document, and there are a total of 500 words in this document. Thus, Term Frequency for "data mining" is 50/500 = 0.1 = 10%.

2.2 Inverse Document Frequency (IDF)

Inverse Document Frequency is a frequency value that measures how important a word is in a collection of documents. When we compute TF, every token that we find in a document is equally important for us. However, some tokens, which we call stop words, such as you, he, she, it and is, appear a lot of times in a document but are less important than other tokens. Therefore, we need a way to scale up the important tokens, and we can achieve that by calculating Inverse Document Frequency (IDF). In this project, we use document frequency for each possible concepts instead of using actual IDF, and then we set a minimum and maximum threshold points to determine if a possible concept is a key concept or a generalized stop word.

The computation of Inverse Document Frequency:

IDF(t) = ln (the total number of documents / the number of documents containing t)

For instance, assuming a token appears in 500 documents and the total number of documents is 500, therefore

IDF(t) = ln(500/500) = 0

IDF(t) is equal to zero. It means that the token is not an important token in the collection of 500 documents.

2.3 Weakness of TF-IDF

The SSE takes a set of documents as input and tokenizes input streams into tokens and records TF and DF to compute TF-IDF for each token. It uses TF-IDF value to decide whether a token is a keyword or a stop word. A token can be regarded as a stop word if its TF-IDF value is nearly equal to zero. However, computing TF-IDF to remove stop words slows SSE significantly. There are two conditions that cause the TF-IDF value to nearly equal zero: the first condition is that a token appears a few times in a document (TF value is very small); the second condition is that a token appears in almost all of the documents (IDF value is nearly equal to zero). The TF value is very easy to determine. But the IDF value is very expensive to compute because we have to scan each word in a collection of documents. Our SSE focuses on indexing all the data from the

World Wide Web. There are billions of webpages on the World Wide Web. If our SSE uses TF-IDF to filter stop words, we have to scan all the billions of webpages and record the TF value and the DF value for each token that appears in all the webpages. After we get TF-IDF values for each token, we use TF-IDF values to decide which tokens can be discarded as stop words. It will take a significant amount of processing time and enormous memory to only generate frequent 1-Itemsets, which are keywords such as {data}, {mining}, {analysis}. Our SSE aims to produce concepts which are frequent finite sequences of keywords (not necessary in a consecutive order in a document). It uses a paragraph of keywords to generate different combinations of keywords to form possible concepts. For example, we have 3 keywords in a paragraph: {data}, {mining}, and {analysis}, and we need to generate {data mining}, {data analysis}, {mining analysis}, which are frequent itemsets with two keywords (2-Itemsets), and also we need to generate {data mining analysis}, which is a frequent itemset with three keywords (3-Itemsets).

We assume that all 30 words are keywords in a paragraph.

the possible 1-Itemsets = $\frac{30!}{1!(30-1)!} = 30$

the possible 2-Itemsets = $\frac{30!}{2!(30-2)!} = 425$

the possible 3-Itemsets = $\frac{30!}{3!(30-3)!} = 4{,}060$

the possible 4-Itemsets = $\frac{30!}{4!(30-4)!} = 27{,}405$

the possible 5-Itemsets = $\frac{30!}{5!(30-5)!} = 142{,}506$

the possible 6-Itemsets = $\frac{30!}{6!(30-6)!}$ = 593,775

the possible 7-Itemsets = $\frac{30!}{7!(30-7)!}$ = 2,035,800

As we can see, the growth of possible n-Itemset is exponential and it quickly becomes an unmanageable problem. That is because SSE's data analysis component (Sutoken program) takes a huge number of comparisons to compute TF-IDF value for each possible concept. Therefore, TF-IDF's drawback for this project is obvious.

3. STOP WORDS

3.1 Introduction of stop words

Stop words are a set of commonly used words that appear many times in a document but have less importance. Stop words are a critical part of data mining and text mining. The simple strategy to determine a stop word is to sort all tokens by collective frequency, which is the number of times each token appears in the document collection, and then remove most frequent tokens to reduce the cost of processing and storing common words.

Stop words are entirely excluded from vocabulary for us to focus on important words. For example, there is a phrase “how to create database.” It contains two stop words “how” and “to.” If search engine tries to use “how,” “to,” “create” and “database” as keywords to find desired HTML pages, it can find a lot more HTML pages that contain “how” and “to” than those that contain “create” and “database.”

Stop words are high frequent words. If we use TF-IDF to remove them, their IDFs are near zero. In SSE, we need to extract keywords and finite frequent itemsets (sequence of keywords) [3], but as mentioned, computing TF-IDF is a very expensive way to handle the job. Therefore, we need to bypass TF-IDF and remove all kind of stop words.

There are 3 classes of stop words: English-grammar-based stop words, Metadata stop words, and Topic-specific stop words.

3.2 First class of stop words: English-grammar-based stop words

English-grammar-based stop words are a set of common English words to express our ideas. English grammar can identify English-grammar-based stop words.
For example:

Determiners: Determiners are used to mark nouns. A noun will usually follow a determiner. An English phrase “the name” contains a determiner “the” which is followed by a noun “name.” There are other determiners such as “a,” “an,” “another.”

Coordinating conjunctions: Coordinating conjunctions are used to connect words, phrases, and clauses such as “for,” “but,” “or,” “yet,” “so.”

Prepositions: Prepositions are used to express temporal or spatial relations such as “in,” “under,” “toward,” “before.”

In order to remove English-grammar-based stop words, we utilize one of the existing English stop word lists that contain all English grammar based stop words.

3.3 Second class of stop words: Metadata stop words

Metadata stop words are a set of stop words which usually appear in HTML pages. For instance, most of the metadata stop words appear in a navigation bar such as “home,” “contact.” The navigation bar is a user interface element within a webpage that contains many links to other sections of the website. It is one of the main components of a website, which means that most webpages within the website display the same navigation bar. This also means that the words contained in the navigation bar will be displayed on most webpages the users are visiting. Those words, we call “Metadata stop words,” affect the efficiency of SSE. We intend to use an HTML parser to remove Metadata stop words.

3.4 Third class of stop words: Topic-specific stop words

The third class of stop words is Topic-specific stop words, which are not common English-grammar-based stop words that can be identified by English grammar or the keywords that we want to capture. Topic-specific stop words usually appear many times in a specific field. For example, tokens such as “mcg,” “Dr.,” “patient” appears mostly in clinical documents. These tokens have a large document frequency but less importance.

4. WEB CRAWLER TO REMOVE METADATA STOP WORDS

Web crawler is a program that automatically crawls HTML pages. It first retrieves HTML pages from the Internet using a set of URLs as sources, then parses and stores those pages in a local system for further analysis [2]. In this project, web crawler stores all HTML pages as text files and feeds them to the Sutoken program, which is the main program of SSE used to process all HTML pages downloaded by web crawler and capture possible concepts.

However, the previous web crawler we used could not filter Metadata stop words. It simply retrieved all text content from HTML pages and let the search engine handle filtering stop words and capturing concepts. Therefore, all stop words that appear in HTML pages can affect accuracy and efficiency of SSE. Our approach to solving this problem is to provide a web crawler to preprocess data and filter Metadata stop words.

4.1 Related Work

Web crawlers are written in many different programming languages to provide services for a variety of purposes. In this section, we describe some well-known web crawlers and the difference between our web crawler and other well-known web crawlers.

Apache Nutch

Apache Nutch is an extensible and scalable open source web crawler on Hadoop. Apache Nutch is written in Java programming language. It aims to index the World Wide Web and provides a highly modular architecture that extends user's customized functionality with the help of some interfaces like Parse, Index, and ScoringFilter. [6]

The workflow of Apache Nutch is showed below:

1) Initialize Crawler Database and inject seed URL
2) Generate fetch list: select URLs from Crawler Database for fetching
3) Fetch URLs from fetch list
4) Parse documents: extract content, metadata and links
5) Update Crawler Database status, score, and signature, and then add new URLs
6) Invert links: Map anchor text to document the links point to
7) Calculate link rank and update Crawler Database scores
8) Delete duplicate documents by signature
9) Index document content, metadata and anchor text

Googlebot

Googlebot is a crawling bot used by Google search. The computer program decides which site should be crawled, how often Googlebot should crawl a certain site, and how many pages Googlebot should crawl. It provides URL extraction and full-text indexing and besides its URL server can handle a large amount of URLs. [7]

WebRACE

WebRACE is a crawler coded in the Java programming language. WebRACE gets requests from users to crawl certain HTML pages. It is notified and starts to re-crawl certain page when the provider changes the page. The most important feature is that WebRACE does not need a seed URL to crawl the World Wide Web. [8]

Differences between our web crawler and other well-known web crawlers

Our web crawler is much simpler than other well-known web crawlers. Our web crawler starts from a seed URL, retrieves an HTML page, parses the HTML page using an HTML parser (modified to suit our requirements), stores parsed HTML data in txt files, and processes another URL. The HTML parser is used to remove Metadata stop words. Our web crawler has very limited functionalities, but it serves our needs. The reason is that we want to test if our web crawler with a modified HTML parser can work well as we expected. Thus, a simple web crawler can fulfill our needs and this project focuses on improving data analysis of our SSE. The primary difference between our web crawler and other well-known crawlers such as Apache Nutch is that well-known crawlers extract and index all data from an HTML page but our web crawler aims only to extract meaningful data by eliminating Metadata stop words.

4.2 Our Study

The original idea of our web crawler is to let the server side handle removing Metadata stop words. We intend to send our HTML parser to a server, and the server uses the HTML parser to remove Metadata stop words, and then the server sends less but more meaningful data back to our web crawler. However, this idea does not work well. Therefore, we decided to crawl HTML page and parse it in the local system.

We want to integrate an HTML parser and TD-IDF into a basic web crawler program to preprocess the data before we feed the data into SSE. In this way, we can make our web crawler smarter to pre-analyze data. The HTML parser can parse HTML files and provide text extraction and link extraction, and also it can remove Metadata stop words, such as “home,” and “contact” in the navigation and footer menu. By using HTML parser, we can get rid of all HTML tags and only extract meaningful text data in HTML files.

4.3 Implementation of web crawler

The high-level architecture of web crawler as shown in Figure 1 below explains the data flow from downloaded HTML pages to local disk. The primary objective of the web crawler is to filter stop words and capture related concepts by using HTML parser and TF-IDF tool. We split the whole work into the following two steps: The first step is to create a web crawler program and then add an HTML parser into it; the second step is that we record TF and IDF for each token to compute TF-IDF.

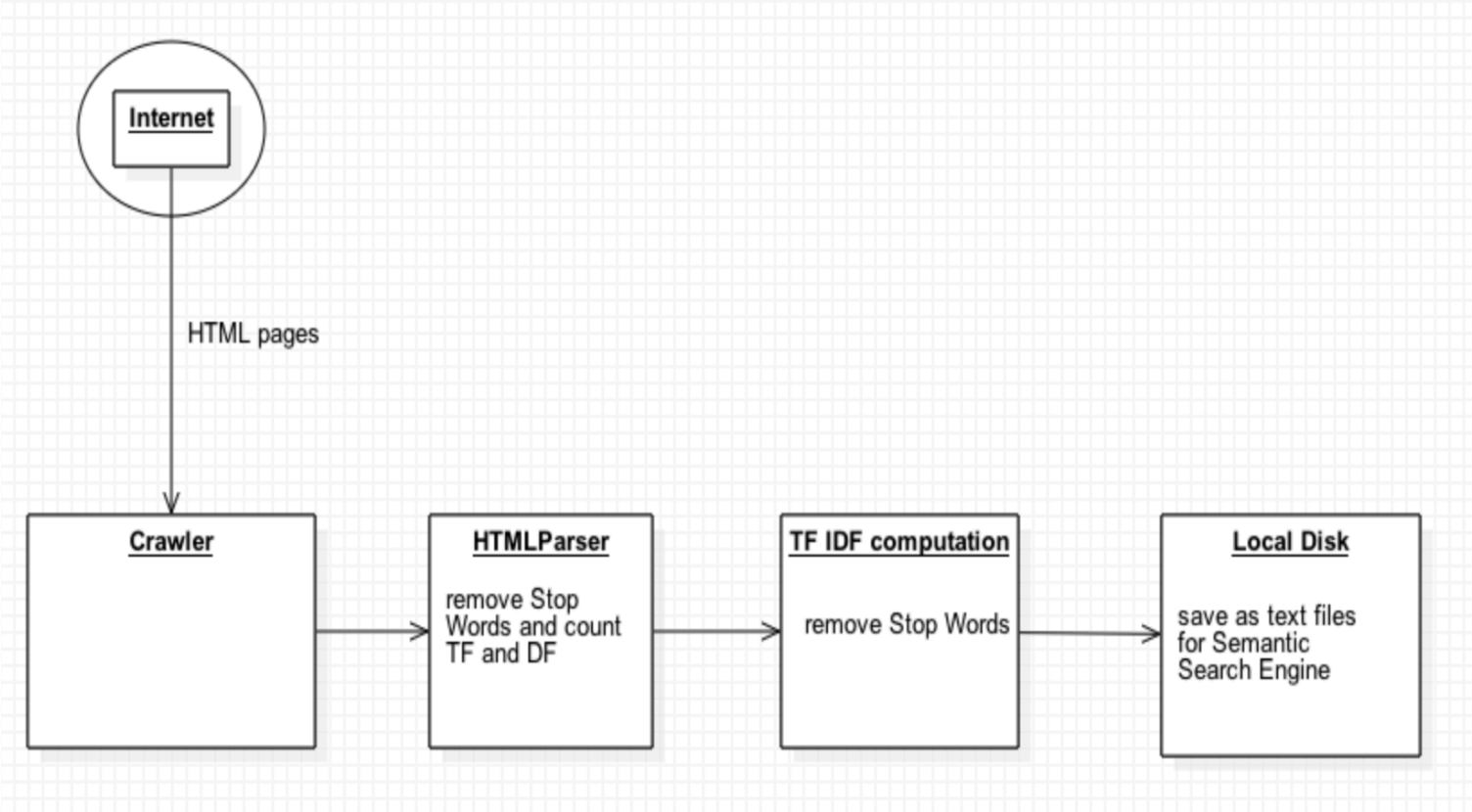

Figure 1. High-Level architecture of web crawler

4.3.1 First step: create a web crawler program and add an HTML parser

The following Figure 2 is a Class diagram for web crawler:

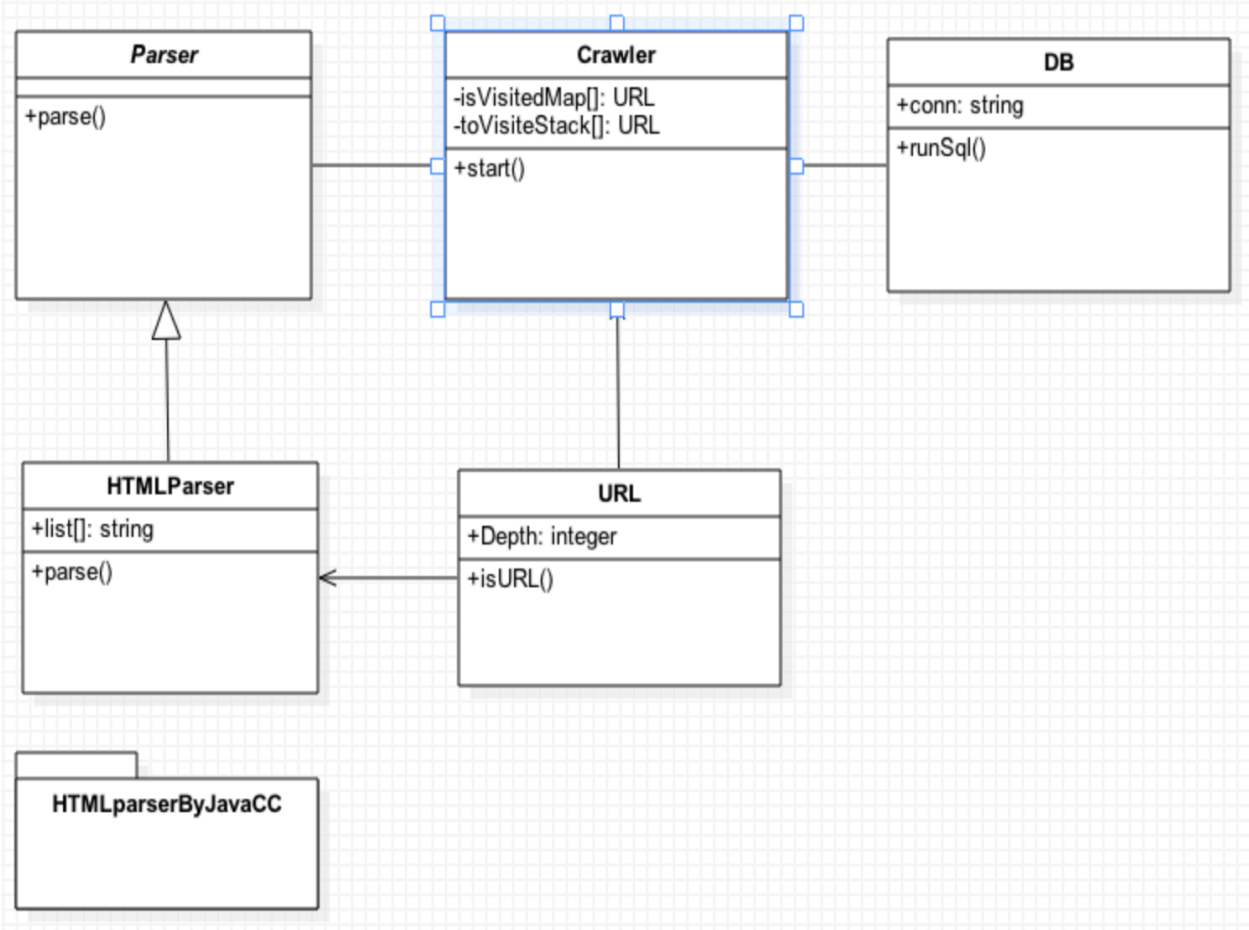

Figure 2. Class diagram for web crawler

Crawler.java

The main functionality of Crawler.java is to collect command line input and set up all required components such as HTML parser. It can start to run the program by using command line input, which is a root URL. It can also check if there are URLs that are not visited by the program. If there are URLs in toVisitStack (a queue data structure which has First In First Out mechanism), it retrieves all data using the first URL in toVisitStack and sends fetched HTML data to HTML parser.

DB.java

The main functionality of DB.java is to set up a database connection and execute SQL statement. We create a Crawler_record database that has two tables. One is isVisitedURL table, which stores all visited URLs. The other table is toVisitedURL table that stores all unvisited URLs.

```
public ResultSet runSql(String sql) throws SQLException {
    Statement sta = conn.createStatement();
    return sta.executeQuery(sql);
}
```

Figure 3. runSql() Function

Parser.java

Parser.java is an abstract class that defines common behaviors, which can be inherited by subclasses. For example, HTMLParser.java extends Parser.java. In this program, we only use HTML Parser, but we can implement more parsers to achieve extended functionalities.

HTMLParser.java

The main functionality of HTMLParser.java is to parse HTML files and collect new URLs. All non-alphabetical characters are removed because our HTML parser can only parse English phrases. HTML parser can remove all HTML tags.

The data we remove from HTML files are any Scripts such as JavaScript, Image, and Multimedia. JavaScript is a programming language that does not contain any

useful textual information. Parsing JavaScript not only affects the accuracy of the search engine but also generates more processing time and memory usage.

Java Compiler Compiler (JavaCC)

JavaCC is an open source parser generator written in Java programming language. It can use formal grammar written in EBFN notation to generate the desired parser. In this project, our web crawler utilizes JavaCC to generate an HTML parser. HTMLparserByJavaCC is a set of java files created by JavaCC which can also be used to create other parsers, such as a PDF parser, to preprocess PDF files. We can integrate more parsers into web crawler to handle different files crawled by web crawler in the future.

URL.java

URL.java is used to create URL objects. The program uses URL Objects to store the Depth of each URL. The Depth variable is used to record how far the crawler goes in Breadth First Search (BFS).

We use Breadth First Search (BFS) algorithm in the web crawler to process URLs. The basic idea of this algorithm is that it starts from the root node and visits all neighbor nodes at the same level. When all nodes at the same level are visited, it goes to the next level and visits all nodes at that level. It stops when there is no next level or the target is found. In this project, BFS suits our situation, and it helps to process

all URLs at the same depth first and then moves to explore all URLs at the next depth.

The algorithm of web crawler:

```
While (toVisitStack is not empty) {
        URL link = toVisitUrlList.pop()     // get next URL to visit
        If (link.depth <= MaxDepth && link.notVisited()) {
                Download HTML file from Internet use Link
                HTML Parser parses HTML file
                Search for new URLs in the downloaded HTML file
                Store new URLs in toVisitList and update depth for new URLs
                Extract text data from HTML files and store into output file
                Store visited URL into database
        }
}
```

Use BFS to process URL:

In the program, we use toVisitStack stack, a FIFO queue, to store new URLs that are fetched from an HTML page. According to BFS Algorithm, the program starts from

the root URL, visits all neighbor URLs, and then moves to the next level of neighbors. The toVisitStack stack is used to supply URLs for the crawler.

Run web crawler with HTML parser:

The following is the text file generated by web crawler. After we run the web crawler program using "www.sjsu.edu" as the root URL, the program generates a text file as a result, which contains only text data from www.sjsu.edu. Based on the result below, we know that the HTML parser successfully removes all HTML tags but the primary objective of the project is not accomplished.

```
www.sjsu.edu
San Jose State University - Powering Silicon Valley | San Jose State UniversitySkip to Main
ContentSJSU Homepage San Jos State UniversitySite NavigationDiscover SJSUAbout
SJSUAcademicsAdministration and LeadershipAthleticsCalendars and EventsInitiatives and
PartnershipsKingLibraryNewsResearchVisit SJSUFuture StudentsMySJSUAcademicsAdmissionsCalendars and
Events FinancesHousing OptionsNew StudentsVisit SJSUCurrent StudentsMySJSUAcademicsAdvisingCalendars
and EventsCampus LifeDiversityFinancesGraduationHealth and WellnessKing LibraryNew
StudentsSafetyTechnologyFaculty and StaffMySJSUAcademicsCalendars and EventsEmployment and
CareerDirectoryDiversityHealth and WellnessKing LibraryResearchSafetyTechnologyAlumni and
CommunityAlumni AssociationEmployment and CareerDo Business with SJSUGive to SJSUHire SJSU
TalentConnectVisit SJSU Responding to the ElectionsThe university will continue to provide support to
community members who seek our help, and will explore opportunities to further enhance campus safety
and security.                    Learn more.                    Students Gain Practical Election
Night ExperienceAs the votes were being counted, graduate students from the School of Journalism and
Mass Communications contributed to one local television stations extensive coverage.
Bay Area Media Turned to SJSU on Election Night 2016These four political science professors shared
their expertise on six TV and radio                    stations, as the nation awaited election
results. Learn more.                    White House Honors SJSU AlumnusPresident Barack Obama awarded
a National Medal of Arts to SJSU alumnus and El Teatro                    Campesino founder Luis
Valdez at the White House (Photo: John Harrington). Read more about the honor.
IndexAcademicsBookstoreCalendarsCareersDirectoryKing LibraryParking & MapsCampus TourEventsSubmit an
Event | More EventsNewsMore News StoriesBright IdeasMartha J. Kanter Visited SJSUFormer U.S. under
secretary of education spoke on student success Sept. 30.More Bright IdeasSpartan PrideMore Spartan
Pride StoriesPower SJSUKoret Foundation Awards GiftSJSU receives $2 million to improve student success
and graduation rates.More Philanthropy StoriesSJSU Links and ResourcesInformation forAlumniCurrent
StudentsDonorsFaculty & StaffFuture StudentsResearchersCollegesApplied Sciences &
ArtsBusinessEducationEngineeringHumanities & the ArtsInternational & Extended StudiesScienceSocial
SciencesQuick LinksA-Z IndexAcademicsBookstoreBudget CentralCalendarsCanvasCareers & Jobs King
LibraryParking & MapsContact usContact FormDirectoryReport a Web ProblemReport a Title IX ComplaintSan
Jose State UniversityOne Washington Square,408-924-10002016Last Modified: Nov 12, 2016
```

Figure 4. Output file generated by web crawler

The main purpose of adding HTML parser is to remove Metadata stop words. However, we can find a lot of Stop Words such as "Homepage," "SJSUAbout," "Events"

and “Life” in this result. If we use this result as an input for SSE, it makes SSE less efficient and accurate.

How to improve HTML parser:

After carefully analyzing the results above, we figure out two ways to improve the HTML parser.

One way is that we need to remove both the header and the footer from an HTML file. Most Metadata stop words appear in those two parts. The HTML header contains a navigation bar, which is a section of a graphical user interface to help the user access information. The navigation bar is an important component of a website, and every website has a navigation bar. Also, the same navigation bar appears multiple times in a website. The HTML footer is the same as the header. The information obtained from footer appears multiple times but is less important. We modify HTML parser so that it does not extract any data from the header and the footer. The simple approach to achieve this is that HTML parser discards the header and the footer, since the header can be identified as HTML tag <header> or <nav>, and the footer can be identified as HTML tag <footer>.

Another way to improve HTML parser is to level each HTML tag in HTML page and only extract text data from the highest leveled tag. HTML can be converted to a tree structure, and the most valuable information usually appears in highest tags.

```
For example, <html>    // the level is 1
                        <body>  // the level is 2
                                < p> the level is 3
                                        I love computer science
                                </p>
                        </body>
                        <div>   // the level is 2
                                <a>  // the level is 3
                                        I am a cat lover
                                </a>
                        </div>
                </html>
```

The highest-level tags are <p> and <a> and thus we extract “I love computer science” and “I like cat” from HTML files. The highest-level tags contain more relevant information in HTML pages. Thus, to mark the level of each tag is a critical component of HTML parser. After we modified the web crawler program and re-ran the program using same root URL “ www.sjsu.edu,” we get the following result:

```
www.sjsu.edu
Responding to the ElectionsThe university will continue to provide support to community members who
seek our help, and will explore opportunities to further enhance campus safety and security.
Learn more.                    Students Gain Practical Election Night ExperienceAs the votes were
being counted, graduate students from the School of Journalism and Mass Communications contributed to
one local television stations extensive coverage.                    Bay Area Media Turned to SJSU on
Election Night 2016These four political science professors shared their expertise on six TV and radio
stations, as the nation awaited election results. Learn more.                    White House Honors
SJSU AlumnusPresident Barack Obama awarded a National Medal of Arts to SJSU alumnus and El Teatro
Campesino founder Luis Valdez at the White House (Photo: John Harrington). Read more about the honor.
IndexAcademicsBookstoreCalendarsCareersDirectoryKing LibraryParking & MapsCampus TourEventsSubmit an
Event | More EventsNewsMore News StoriesBright IdeasMartha J. Kanter Visited SJSUFormer U.S. under
secretary of education spoke on student success Sept. 30.More Bright IdeasSpartan PrideMore Spartan
Pride StoriesPower SJSUKoret Foundation Awards GiftSJSU receives $2 million to improve student success
and graduation rates
```

Figure 5. The result of web crawler

We achieve the improved result above in two ways. After discarding the header and footer in an HTML file and extracting data from the highest tags, the HTML parser removes most Metadata stop words.

We did an experiment to measure the performance of SSE. We crawled 500 webpages from www.sjsu.edu using an existing web crawler. Then, we crawled the same size webpages from the same website using our web crawler. We used two sets of crawled data to run Sutoken program (the core engine of SSE). The results are shown in Table 1 and Table 2.

Table 1: Result for existing web crawler

Total number of tokens	**268589**
Total processing time (second)	**4224.8490**

Table 2: Result for our web crawler

Total number of tokens	**114540**
Total processing time (second)	**2383.2040**

After we had compared the two results, we found that the total processing time of our web crawler is far less than the total processing time of the previous web crawler. The main reason is that our web crawler has an HTML parser to remove Metadata stop words. The data crawled by the existing web crawler contains many Metadata stop words, and SSE needs to spend more processing time to filter them.

4.3.2 Second step: Add TF-IDF tool

The next step is to add TF-IDF into the program. The main purpose of adding TF-IDF tool is to filter stop words and look for possible concepts. [1]

However, using TF-IDF tool in web crawler is an inefficient and costly way to filter stop words and identify keywords. We need to record TF and DF for each token to compute TF-IDF. When the number of tokens increases, the required memory size will grow rapidly to store all records of TF and DF.

Also, web crawler is designed to crawl a large set of HTML pages. In this situation, counting Document Frequency for all tokens is impossible and very expensive job for a web crawler. A web crawler requires enormous memory and computing power to handle such a huge number of comparisons. Therefore, adding TF-IDF is a not good approach for web crawler.

5. UNDERSTAND TOPIC-SPECIFIC STOP WORDS AND CREATE A STOP WORD LIST:

5.1 Our approach

One way to get Topic-specific stop words is to use a web crawler to crawl a large set of HTML pages as input files. The web crawler extracts all texts from one HTML page and stores them as a text format file (.txt file). Then, we feed files that contain all text data to our SSE.

Our SSE utilizes TF-IDF tool to filter Stop Words. Therefore, we can run the Search Engine and keep recording Topic-specific stop words (High document frequency and IDF is near zero), which are filtered by TF-IDF tool. In this way, we can collect Topic-specific stop words to create a Stop Words List. However, HTML pages lack possible concepts and a word that is not a stop word could be filtered as a stop word. Topic-specific stop words, which are gathered by search engine using HTML pages as input, cannot meet our needs.

Another way is to download 10000 documents that are related to Data Mining and feed those documents to SSE. The 10000 documents are research papers downloaded from IEEE. Data mining research papers contain related and intended concepts and also more suitable and reliable Topic-specific stop words, we call “Data Mining stop words.”

The following steps can illustrate the overall approach to creating a Stop Word List:

1. Download 10000 documents that are related to Data Mining topics. The reason we use Data Mining topics is that we already know the content of those Data mining papers. Based on that, we can easily verify the results of SSE and identify Topic-specific stop words.

2. Convert PDF files to TXT files. 10000 documents are PDF files, while our SSE requires TXT files as input. Dr. Lin’s former students had already completed the first two steps to download 10000 documents and convert them to txt files. It saved us a lot of time and energy to collect those documents.

3. Feed documents to our SSE (Sutoken program). To prepare for feeding all documents to SSE, we did an experiment to measure the search engine’s performance. We ran the Sutoken program for several times, supplied it with a small number of documents each time and counted its processing time. We prepared five collections of documents: 100 documents, 200 documents, 300 documents, 400 documents and 500 documents.

Figure 6 shows that search engine's processing time will grow exponentially when the number of input files increases.

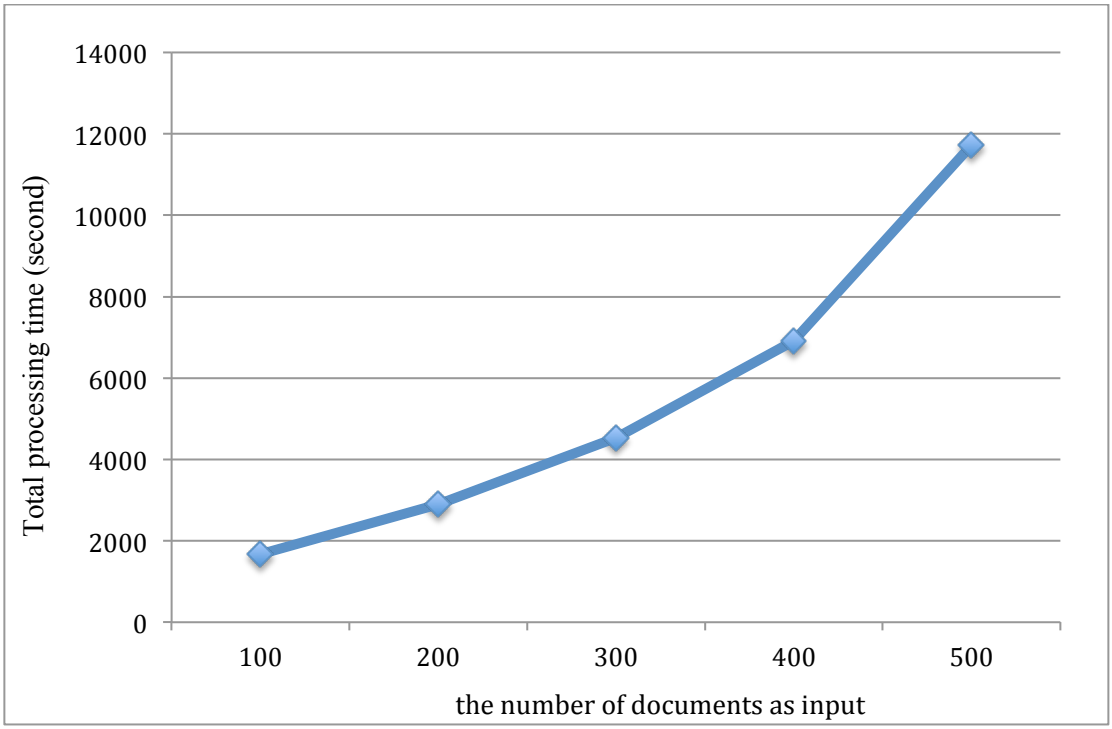

Figure 6: The number of document vs. the processing time

The reason why the processing time grows non-linearly is that our SSE needs to record TF and DF for each token in order to compute TF-IDF, which means when SSE gets a new token then it goes through all other documents and compares with each token in order to get the number of documents containing a specific token. The comparison of each token takes most of the processing time and slows down the program. If we use one computer to process all 10000 documents, it would be an impossible mission to finish processing them in a reasonable amount of time.

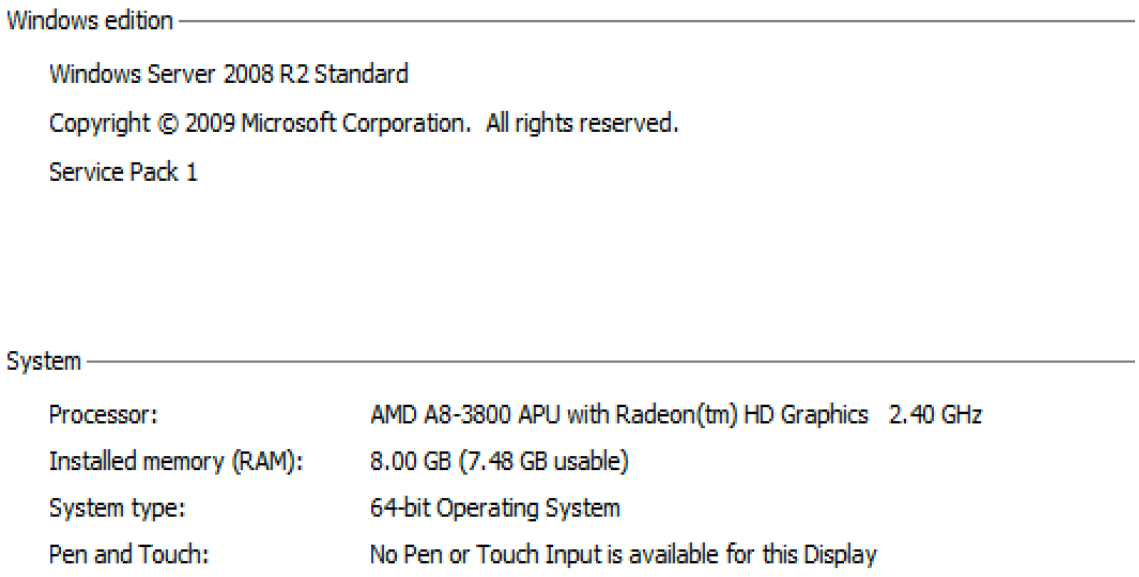

Figure 7. System property

Figure 7 above shows that the computer we are using installs Windows Server 2008 R2 standard and it has a processor that is AMD A8-3800 APU with Radeon(TM) and 8GB memory. This is a standard computer with limited power to handle the analysis of such a large set of data. To process 500 documents, SSE runs 11,718.12 seconds, which is almost 3.26 hours. If we have a more powerful computer, the processing time can be reduced. In the future, we may consider Cloud Computing for its powerful computing technology.

Our approach for processing all documents is to divide them into batches, assign one batch to one student and then each student runs search engine program (Sutoken program) on their computer. There were 60 students who helped us to collect data. We divided them into 20 groups and assigned 500 documents to each group. Moreover, each group was required to use only one computer to run the search engine program with all

500 documents as input files. Thus, we had all 20 groups to work in parallel and run the same search engine program with the same size data. It saved us much time compared with using one computer to process all 10000 documents.

Table 3 contains the top 50 concepts with three token counts captured by one of the 20 groups:

Table 3. Top 50 concepts with 3 token count

ID	Tokens	TokenCount	Frequency	DocFrequency	TokensOrigin
678473	ab ac ad	3	14	6	abe, ace, ade,
741072	ab ac bc	3	19	4	ab, ac, bc,
525038	abstract algorithm associ	3	6	6	Abstract algorithm association
572312	abstract algorithm frequent	3	13	10	Abstract Algorithm frequent
621288	abstract algorithm item	3	6	5	Abstract algorithm, item
525042	abstract algorithm rule	3	5	5	Abstract algorithm rules
621291	abstract algorithm set	3	4	4	Abstract algorithm, set,
775592	abstract apriori algorithm	3	11	7	Abstract Apriori algorithm
525043	abstract associ algorithm	3	11	8	Abstract association algorithm
517021	abstract associ data	3	5	4	Abstract association data
621295	abstract associ item	3	7	6	Abstract association item
517022	abstract associ rule	3	36	23	Abstract association rules
550950	abstract databas rule	3	5	4	Abstract databases rules
621301	abstract frequent algorithm	3	5	4	Abstract frequent algorithm
601844	abstract frequent data	3	9	7	ABSTRACT Frequent data
621302	abstract frequent item	3	14	10	Abstract frequent item
601847	abstract frequent itemset	3	11	9	ABSTRACT Frequent itemsets
749022	abstract frequent pattern	3	11	6	Abstract frequent patterns
621305	abstract frequent set	3	14	11	Abstract frequent set,
525082	abstract import data	3	6	5	Abstract important data

890823	abstract item associ	3	5	4	Abstract item associations
662737	abstract item data	3	6	4	abstraction, item Data
963661	abstract item rule	3	6	4	Abstract items rules
621311	abstract item set	3	14	10	Abstract item set,
601854	abstract itemset data	3	5	4	ABSTRACT itemsets data
525089	abstract larg databas	3	4	4	Abstract large databases
662747	abstract paper algorithm	3	10	7	abstraction. paper, algorithm
749024	abstract paper frequent	3	6	5	Abstract paper, frequent
509115	abstract paper present	3	4	4	Abstract paper presents
646406	abstract problem frequent	3	6	4	Abstract problem frequent
700151	abstract propos frequent	3	5	4	Abstract proposed frequent
525100	abstract rule algorithm	3	11	8	Abstract rules algorithm
517026	abstract rule data	3	5	4	Abstract rules data
525107	abstract rule import	3	6	4	Abstract rules important
621335	abstract rule item	3	8	6	Abstract rule item
662781	accuraci data set	3	12	5	accuracy data sets
706314	accuraci frequent itemset	3	18	4	accuracy frequent itemsets
848847	achiev associ rule	3	12	4	achieved Association rule
700176	acknowledg author thank	3	4	4	Acknowledgement authors thank
532810	acknowledg nation foundat	3	8	4	Acknowledgements National Foundation
509147	acknowledg nation natur	3	7	5	Acknowledgements National Natural
532813	acknowledg nation scienc	3	6	4	Acknowledgements National Science
509155	acknowledg natur china	3	7	4	Acknowledgements Natural china
532816	acknowledg natur foundat	3	10	5	Acknowledgements Natural Foundation
532819	acknowledg natur scienc	3	10	6	Acknowledgements Natural Science
532829	acknowledg paper support	3	5	5	Acknowledgements paper supported
630208	acknowledg research support	3	6	6	Acknowledgments. research supported
532833	acknowledg scienc foundat	3	14	7	Acknowledgements Science Foundation
532839	acknowledg support	3	8	5	Acknowledgements supported

	foundat				Foundation
532842	acknowledg support natur	3	7	6	Acknowledgements supported Natural

The Tokens column represents all concepts captured by the SSE, and each concept contains multiple tokens. For example, “Abstract Apriori algorithm” concept consists of “Abstract,” “Apriori” and “algorithm,” and we group 3 tokens together to capture a concept in the human mind. The TokenCount indicates the number of tokens in a concept. The DocFreq is not a general Document Frequency. It stores the number of documents that contain a specific token/concept. The TokensOrigin records the Original token because the tokens stored in Token column are stemmed using Porter stemmer algorithm [4].

The expected results should include concepts related to Data Mining because the documents we feed are papers related to the topic of “Data Mining.” As we can see, concepts obtained by SSE contain “Apriori algorithm” and “association rules,” two key terms of data mining. However, we also find concepts that contain “ab,” “ac” and “bc.” Those un-meaningful tokens are stop words that survive the TF-IDF test iteration. The TF-IDF tool cannot filter out all stop words, and it only reduces candidate space. We need a smarter way to filter those stop words that survive the TF-IDF test iteration.

The following table contains a small part of Stop Words filtered by TF-IDF test. The SSE, as mentioned before, uses TF-IDF tool to filter Stop Words. We can collect potential Data Mining stop words (High document frequency) filtered by TF-IDF test and add to our Stop Words List. We can simply modify the program to store all potential

Data-mining stop words into a file instead of discarding them. Then, we compare those potential Data-mining stop words with English-grammar-based stop words to reduce candidate space. The reason is that English-grammar-based stop words also have high document frequency and can be filtered by TF-IDF.

Table 4. A part of Stop Words filtered by search engine

removing [stockholm]
removing [stockjobb]
removing [stockmey]
removing [stoica]
removing [stomach]
removing [ston]
removing [stood]
removing [stoplist]
removing [stopword]
removing [storaf]
removing [storedataitem]
removing [storytel]
removing [stouch]
removing [stq]
removing [stra]
removing [straggler]
removing [straighten]
removing [strand]
removing [stranieri]
removing [strategyminmaxload]
removing [straw]
removing [strawman]
removing [strcount]
removing [strcutur]
removing [strdmdr]
removing [streak]
removing [streambas]
removing [streamglob]
removing [streamkdd]
removing [streammzn]
removing [streamt]
removing [streetart]

removing [streetmap]
removing [strehk]
removing [strehl]

4. After we collect all Data-mining stop words that are filtered by search engine, we start to execute Map Reduce. Map Reduce is a program model that can process and generate a large data set.

5.2 Map part:

In the Map part, we need to map the same concepts together to create a large data set. After we collected all the concepts captured by the 20 groups, we had a total of 20 datasets because each team ran the search engine program on their computer. The goal was to map 20 datasets into one large dataset. Our approach is to compare tokens of each concept. If two concepts have the same tokens, we can map them together.

Before we started to do the Map, we needed to export all data from the database. The search engine program use database to store all captured concepts and URL links (we do not use HTML pages as input. Therefore URL links contain the directories of all documents). There are three tables containing all data:

1) Concepts table stores all concepts and information about those concepts
2) URLPages table stores all URL links but now it contains all directories of documents
3) MapConceptsToURL table is used to map concepts to URLs

We asked each group to export their three tables into 3 flat text files: Concepts.txt, URLPages.txt and MapConceptsToURL.txt. Then we collected all exported files and started to do the Map.

Figure 8 shows the code for mapping concepts. It opens each concepts.txt file and reads its content row by row. Then, it will check concept_index map to see if this concept has shown before. If it has, we simply update the existing concept row in final_concept_rows array by adding the Frequency and DocFrequency of current processing concept to the Frequency and DocFrequency of existing concept row. If this concept shows for the first time, we insert a new concept row into final_concept_rows array. We also need to generate a new ID for each concept instead of using the old ID which came from students' data to prevent a collision.

```
with open(concepts_path) as f:
    reader = csv.reader(f, delimiter='\t')
    i = 0
    for row in reader:
        if i > 0 and len(row) > 4:
            concept_token = row[1]
            if concept_token in concept_indexes:
                concept_index = concept_indexes[concept_token]
                existing_row = final_concept_rows[concept_index]
                existing_row[3] = int(existing_row[3]) + int(row[3])
                existing_row[4] = int(existing_row[4]) + int(row[4])
                concept_id_reassign_map[row[0]] = concept_index
            else:
                concept_index = len(concept_indexes)
                concept_indexes[concept_token] = concept_index
                concept_id_reassign_map[row[0]] = concept_index

                dup_row = list(row)
                dup_row[0] = str(concept_index)
                final_concept_rows.append(dup_row)

        i += 1
```

Figure 8. The code for mapping concepts

Figure 9 showed below opens each URLPages.txt file and starts to read each row of data. We simply create a new ID for URL row data and insert the URL row into a final_urlpages_rows array. The reason we create new ID for each URL row is also to prevent collision. We do not need to update URL row because URLPage.txt contains the directories of documents and the directory cannot be the same for each group.

```
with open(urlpages_path) as f:
    reader = csv.reader(f, delimiter='\t')
    i = 0

    for row in reader:

        # ID regeneration; same with concepts
        urlpages_id_reassign_map[row[0]] = urlpages_index

        if i > 0 and row[0] != '0':
            dup_row = list(row) # duplicate the row
            dup_row[0] = str(urlpages_index)
            final_urlpages_rows.append(dup_row)

        i += 1
        urlpages_index += 1
```

Figure 9. The code for get URL pages

Figure 10 showed below opens each MapConceptsToURL.txt file and read each row of data. We need to use old concept ID and old URL id to find reassigned concept ID and URL id. Then, we insert new MapConceptToURL row data with new concept ID and new URL id. Last, we generate new Concepts.txt, URLPages.txt, and MapConceptToURL.txt and start to import all concepts data using these three files.

```
with open(mctu_path) as f:
    reader = csv.reader(f, delimiter='\t')
    i = 0

    for row in reader:
        # we want to make sure the MapConceptToUrl has the right ID
        # references since we regenerated them
        concept_in_map = row[0] in concept_id_reassign_map
        urlpages_in_map = row[1] in urlpages_id_reassign_map

        if i > 0 and row[0] != '0' and concept_in_map and urlpages_in_map:
            reassigned_concept_id = concept_id_reassign_map[row[0]]
            reassigned_urlpages_id = urlpages_id_reassign_map[row[1]]
            final_mctu_rows.append([reassigned_concept_id, reassigned_urlpages_id])
        else:
            pass

        i += 1
```

Figure 10. The code for reassigning ID

When we do the map part, we discover that most tokens that appear in all 20 data sets are intended concepts we are looking for and those tokens that have high document frequency and do not appear in all 20 data sets can be regarded as potential Data-Mining

stop words. We need to check if those potential Data-Mining stop words are really stop words.

5.3 Reduce Part:

Reduce Part is to let the human mind check if those potential Data-Mining stop words are really stop words. We have already collected some of the Data-Mining stop words that are filtered by TF-IDF test. However, there are still a lot of Data-Mining stop words that remain in concepts. TF-IDF cannot help us identify those stop words hiding in concepts. Thus, we decided to let humans read the concepts within the short paragraphs where we found these concepts to help us check the results.

60 students participated in the Reduce part. We randomly assigned 300 concepts to each student and provided the Token Original and a short paragraph, where we found the concept. Figure 11 shows how the Reduce part works:

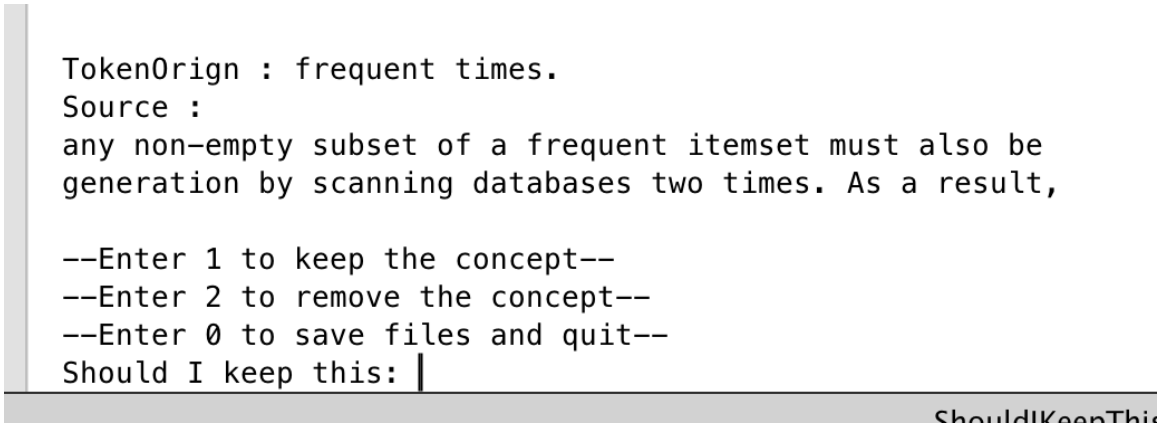

Figure 11. Screen shoot for a Simple program

We created a simple program and let a student read the Token Original and Source and then decide whether we should keep the concept or not. If the student decided that we should not keep the concept, it means that the concept contains unfiltered stop words and then we let the student decide which words should be stop words. The tokens we provided for all students were high frequent tokens, i.e., those that have high document frequency. If the student decided that we needed to keep the concept, it meant that it was an intended concept.

After collecting all the reports of the Should I Keep This Concept program, we got 1584 Data-Mining stop words. Table 5 shows the top 100 of them that are found by using the human mind.

Table 5. A part of Data-Mining Stop Words filtered by human mind

confer	487
sigmod	440
acm	422
set	370
larg	363
proceed	310
intern	308
item	286
pp	177
srikant	164
agraw	160
imielinski	160
proc	136
comput	134
discov	131
support	131
washington	128
scienc	114
engin	111

5. The last step is to collect all Data-mining stop words and create a final Stop Word List. We had already collected a part of Data-Mining stop words that are filtered by TF-IDF test before. Then, we collect a part of Data-Mining stop words that survive the TF-IDF but were identified by the human mind. We combined two sets of Data-Mining stop words and created a Stop Word List.

5.4 Understand the Topic-specific stop words

English grammar cannot identify Topic-specific stop words. As we mentioned, they appear mostly in a specific field, and the common English-grammar-based stop words list does not cover such Topic-specific words. We needed to do the experiment on 10000 documents about “data mining” to collect Data Mining stop words, which can be used to filter the stop words that are hiding in the possible concepts. In the future, we hope to find the patterns of Topic-specific stop words and create different domain stop word lists for different topics or fields.

5.5 Re-run Search Engine with new Stop Word List and compare results

Inserting Stop Word List is straightforward. Our SSE uses the database to store all stop words in a StopWord table. All we need to do is to add all stop words into this table. Our SSE will filter all stop words in a document using the table to find matches. Table 6 shows a part of the StopWord table.

Table 6.StopWord table

	StopWord
1	
2	a
3	abl
4	about
5	abov
6	accord
7	accordingli
8	across
9	addition
10	afl
11	after
12	afterward
13	again
14	against
15	ahead
16	ak
17	ala
18	all
19	almost
20	alon
21	along
22	alreadi
23	also
24	although
25	alwai
26	am
27	among
28	amongst
29	amoungst
30	amount
31	an

After we inserted Stop Word List into the SSE, we ran the Sutoken program using the same system and the same input documents: 100 documents, 200 documents, 300

documents, 400 documents and 500 documents. Figure 12 shows the results of this experiment.

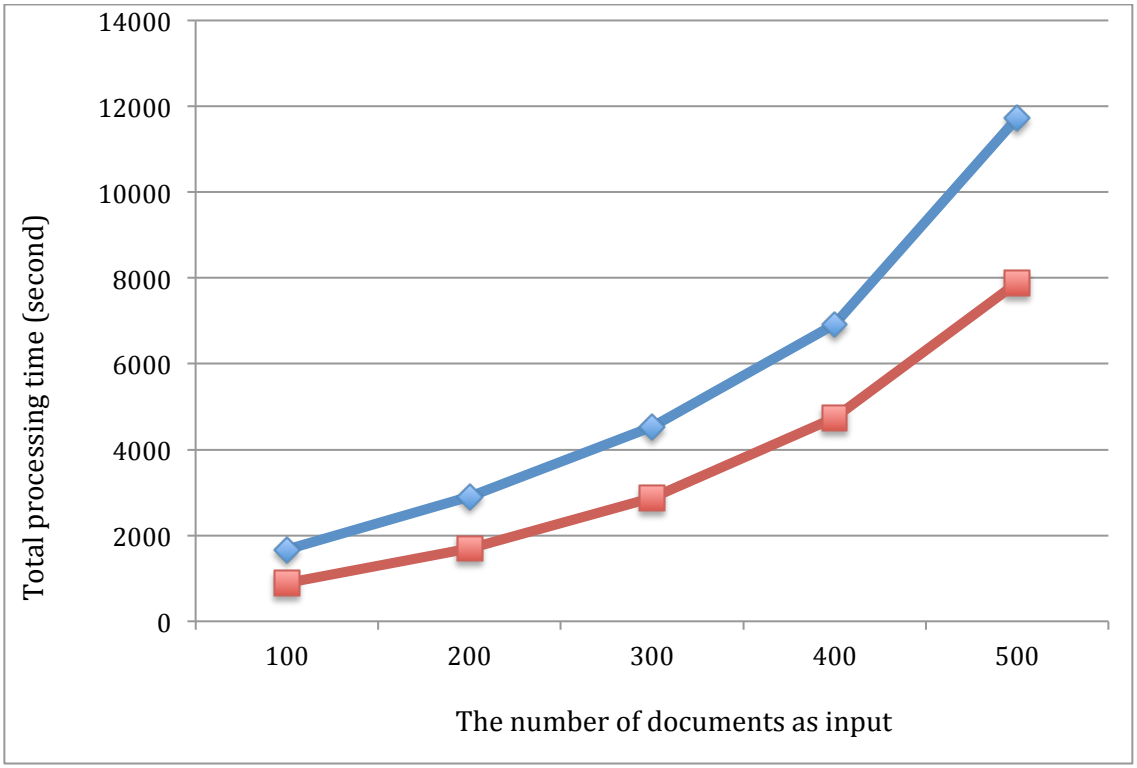

Figure 12. Comparison of the results

The blue line represents the result of the previous experiment. The red line represents the result of this experiment. It is evident that using a new Stop Word List can significantly reduce the processing time of SSE. By using Stop Word List, the SSE only need to remove all kind of stop words once, and then it can focus on finding more suitable and relevant frequent itemsets. Therefore, we can achieve the objective of this project.

6. CONCLUSION:

In this project, we classified three classes of stop words: English-grammar-based stop words, Metadata stop words, and Topic-specific stop words. We created a web crawler program and added a modified HTML parser into it to discard Metadata stop words and reduce the processing time significantly for SSE.

We successfully generated a large set of data and brought human effort into this project. We used MapReduce to collect Topic-specific stop words from 10000 documents about data mining. In the end, we created a final Stop Word List and inserted it into the SSE. Moreover, we re-did the first experiment with the same conditions, but this time our SSE used our final Stop Word List to filter Data-mining stop words. The result of the second experiment shows that our approach improves the SSE's efficiency, and we have achieved this project's objective.

REFERENCES

[1] Tsau Young (T. Y.) Lin, Albert Sutojo and Jean-David Hsu: Concept Analysis and Web Clustering using Combinatorial Topology (2006)

[2] Christopher Olston and Marc Najork: Web Crawling, http://infolab.stanford.edu/~olston/publications/crawling_survey.pdf

[3] Tsau Young (T.Y) Lin: Attribute (Feature) Completion – The Theory of Attributes from Data Mining Prospect

[4] http://nlp.stanford.edu/IR-book/html/htmledition/normalization-equivalence-classing-of-terms-1.html

[5] Manning, C., Raghavan, P., & Schutze, H: Introduction to Information Retrieval (2008) Cambridge University Press.

[6] Rohit Khare, Doug Cutting, Kragen Sitaker, Adam Rifkin: Nutch: A Flexible and Scalable Open-Source Web Search Engine, http://commerce.net/wp-content/uploads/2012/04/CN-TR-04-04.pdf

[7] https://en.wikipedia.org/wiki/Googlebot

[8] Demetrios Zeinalipour-Yazti, Marios Dikaiakos: Design and Implementation of a Distributed Crawler and Filtering Process