Crawling on the World Wide Web

Li Wang
Virginia Tech
Liwang5@vt.edu

Edward A. Fox
Virginia Tech
fox@vt.edu

ABSTRACT

As the World Wide Web grows rapidly, a web search engine is needed for people to search through the Web. The crawler is an important module of a web search engine. The quality of a crawler directly affects the searching quality of such web search engines. Given some seed URLs, the crawler should retrieve the web pages of those URLs, parse the HTML files, add new URLs into its buffer and go back to the first phase of this cycle. The crawler also can retrieve some other information from the HTML files as it is parsing them to get the new URLs. This paper describes the design, implementation, and some considerations of a new crawler programmed as an learning exercise and for possible use for experimental studies.

1. Introduction:

As the size of the Web grows rapidly, building a web search engine for the World Wide Web becomes the interest of many computer scientists. With a web search engine, the ability for people to surf the web and locate a specific page is increased.

A web crawler is an important module in a search engine system. It navigates through the WWW according to its algorithms and downloads the pure files onto the local hard drive. Other modules, like an Indexer and Query Engine, then process those data and provide refined information. Hence the quality of the web crawler directly affects the quality of the subsequent steps. Since the Web is so huge, it's impossible for any crawler to download all the pages available on the Web. There are different estimates of the size of the WWW reported in different papers, but most of them agree that there are more than a billion pages on the Internet. On average, the size of a web page is about 5-10 KB. Multiplying these two numbers together indicates that there are at least several terabytes of HTML files on the Web. And those web pages are not static; they are changing. To guarantee that the downloaded collection represents the most updated one, the crawler needs to remember all the pages it has downloaded, trace their changes and replace the ones that have expired. Hence, time efficiency is the most important characteristic of such web crawlers. If the crawler is not fast enough, the collection on the local hard drive will be only a small subset of the WWW. Then the search engine will not provide high quality searching results for the users, since it is not aware of the existence of other popular and more attractive web pages.

Hence the web crawler I made concentrates on time efficiency. The program is coded in Java and connects to a MySQL database. You may ask why I choose Java instead of C++ or other non-interpreted language. Of course, a program written in those languages runs faster than a program written in Java, especially after the optimization compilation. However the crawler is not a CPU-intensive program, but an IO-intensive program. It will spend significant time on network operation and hard drive writing, instead of intensive computing. The running time difference between a Java program and a C++ program will not significant in such types of programs. That's why Java programs are running on more and more web servers. And Java has a more important feature, platform independence. That is the real reason why the crawler is written in Java. Platform independence implies that the crawler can be deployed to any platforms that support Java, without modification. If the crawler's speed of page downloading doesn't satisfy your requirement, you can simply copy the program to another machine and run the crawlers simultaneously without any source code modification or recompilation. If they are accessing the same database, you don't need to worry about redundancy problems; the collections downloaded by each crawler program will be distinct and unique. The overall performance will be increased significantly. The peak crawling speed when the crawler is running on my personal computer (256M RAM,

667MHz CPU, Ethernet) can reach 30Mbytes per minute if I deploy two crawlers on it. If you have enough machines and network bandwidth, you can get whatever speed you want.

2. Program Framework:

- Basic Skeleton:
  The overall structure is shown in Figure 1:

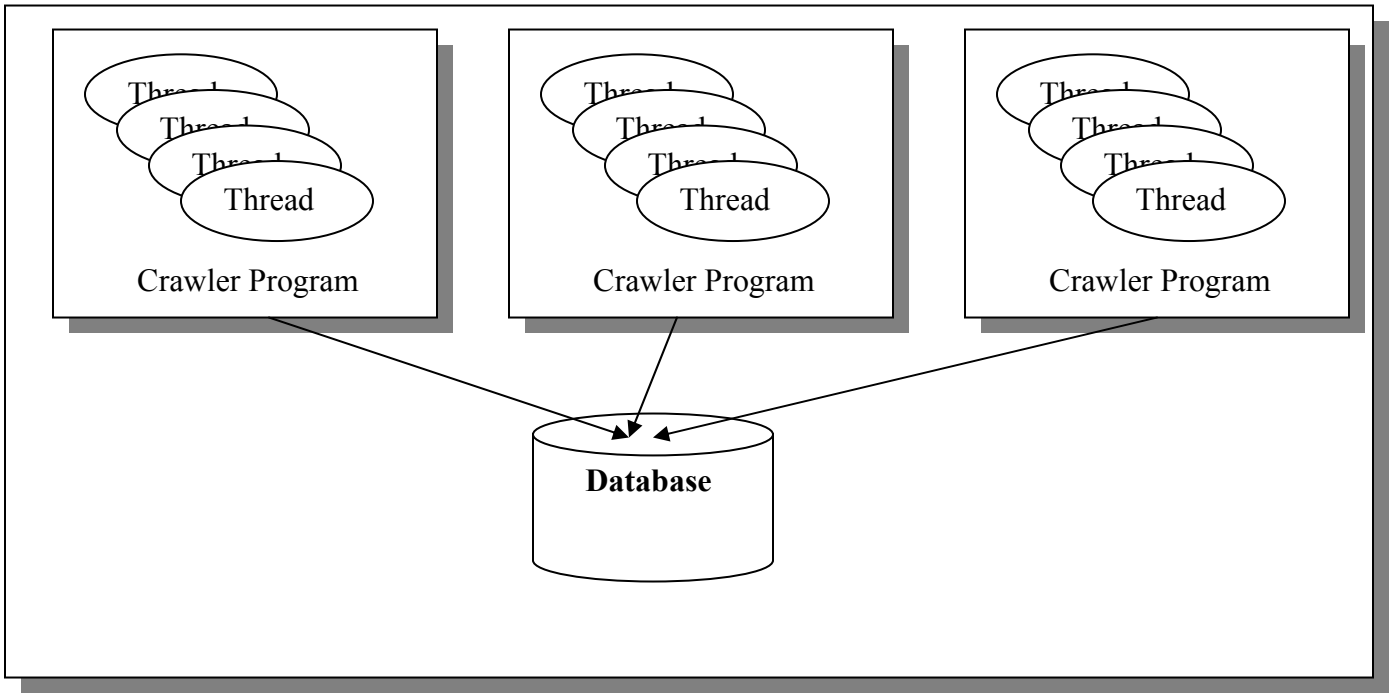

Figure 1. Overall structure

Each program involves many threads. The threads are simultaneously running within the crawler program. The crawler programs also are running simultaneously, downloading the different domains on the Internet according to the configuration file. The synchronization of threads is controlled by the NewUrlBuffer module.

- Threads:

You can deploy many crawler programs or different machines to speed up the downloading speed. Each crawler program includes many simultaneous threads, the number of which is specified in the configuration file conf.crawler. The configuration file controls the parameters for the whole program. Since there are many parameters affecting the performance of the web crawling, it's necessary to control those factors

through a separate file, instead of hard wiring them into the source code. And in this way it's easy to observe the performance differences with different parameters; we don't need to compile the source code again and again. The configuration file will be mentioned later.

All the threads are uniform, running the same code. Figure 2 shows a simplified view of the actions included in a crawler thread.

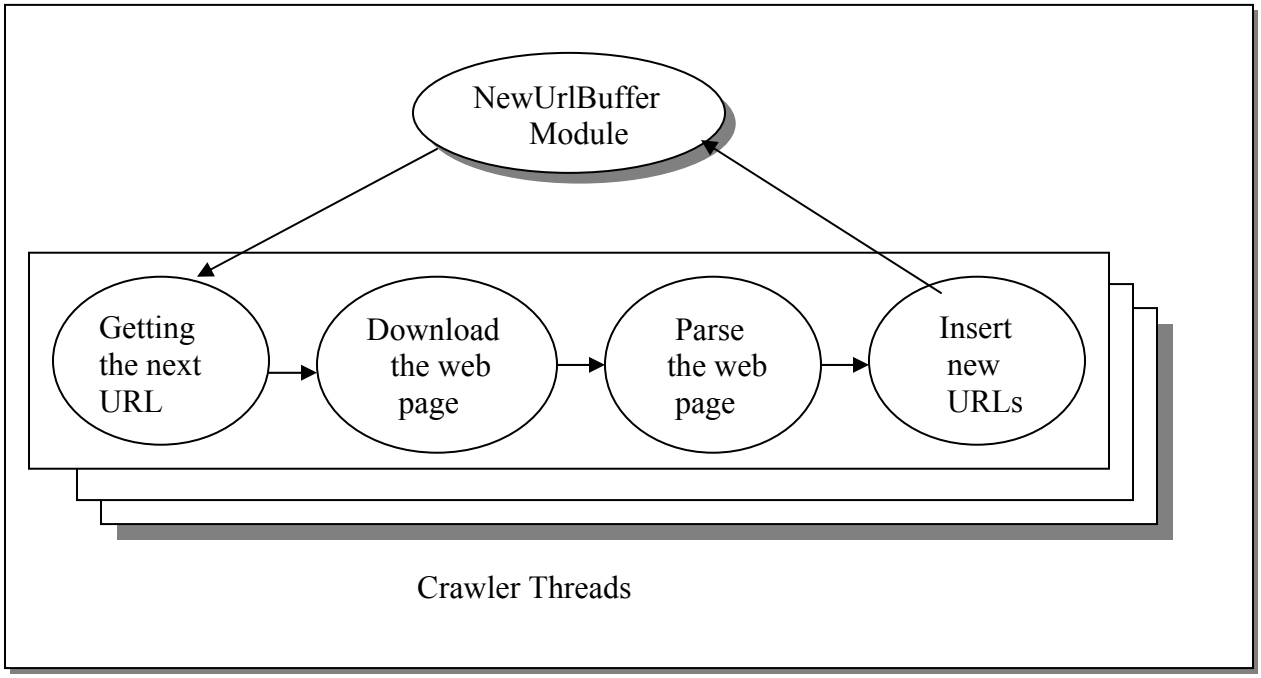

Figure 2. Crawler thread actions

All of the actions for a crawler thread are included in the file CrawlerThread.java. The NewUrlBuffer module is included in NewUrlBuffer.java file.

Each thread creates a new Parser in the thread's constructor. The Parser is included in Parser.java. Its major function is to retrieve the useful information, like URLs, linkage information, meta-information in the HTML text, etc. The parser will be mentioned later. First, the thread will try to get a new URL from the buffer (NewUrlBuffer). If some URLs are available, it will prepare to download the HTML file from the remote server, otherwise it will sleep for several seconds and then try to get a new URL again. It is only possible for threads to sleep in the early phase of the program, in cases when there are not enough seed URLs provided by the user in the Buffer. After the program runs for a few seconds, the URL buffer will be jammed with new URLs. In my experiment for HTML file crawling, there are around 10 new URLs in each web page on average. Hence, except during the earliest phase, all the threads are kept very busy downloading and parsing.

After the thread gets a new URL, it will contact the remote web server and get the header fields before it actually begins downloading. The most important thing the header fields can provide is the information about content type and size. The threads can make a decision whether or not to download this file, according to this information. If the content is a program or an image, and the user chooses to download only pure text files, this file will not have been downloaded by the thread.

Now the thread has made the decision to download the file from the given URL. In the current version, for pure text files the thread will first construct a vector and store all the content line by line into this vector. A vector in Java is a dynamic array. What the thread does is to write all the content into the memory first, then pass the reference of the vector to the parser, finally write to the disk. In one of the previous versions, I tried to parse the content at the same time as downloading the file. And in another version, the program saves the file into the hard drive when it is downloading it and parsing it. But for the cases of our observation, the current version has the best performance. There are 7 versions of this program now. In each version, I tried to use different structures and observe their performance. Regarding the different performances with different structures, I will mention them later.

After the thread completely writes the content into memory, it will pass the reference to the parser. According to the configuration file, the parser retrieves the necessary information, including new URLs and linkage information, from the web page. At the same time, the thread stores the content of the URL into the hard drive and writes the new URLs to NewUrlBuffer, which holds all the new URLs.

Now the thread has completed its cycle, and will begin another new cycle.

- Parser:

The parser program is included in Parser.java. Parser is an important module in the crawler program. It needs to correctly retrieve the required information from the HTML file. And its performance is a major determiner of the program performance. To get the new URLs and linkage information correctly is not an easy job. For HTML, the rules for writing tags are not as restrictive as for XML. For example, a programmer can write nested tag " <tagA> <tagB>Information for A </tagA> Information for B </tagB> " in HTML. And he can leave an opening tag without a closing one. A good parser should handle these different cases gracefully and quickly, which means that it should not be too simple or too complicated.

In the current version, the HTML files are treated as separated strings, and the functions associated with String in Java are used, for example, StringTokenizer class, and functions provided by the String class. In a previous version, the HTML files are treated as single characters, and no functions in the String class are used. The parsing process

proceeds character by character. Unfortunately, through measuring its performance, this method is shown to be much slower than the current one, and it is discarded.

The current parser is built as a Deterministic Finite state Automata (DFA). It uses different states to remember the current parsing stage. It also needs to define the input strings to change the state. Figure 3 shows a simplified DFA to get new URLs.

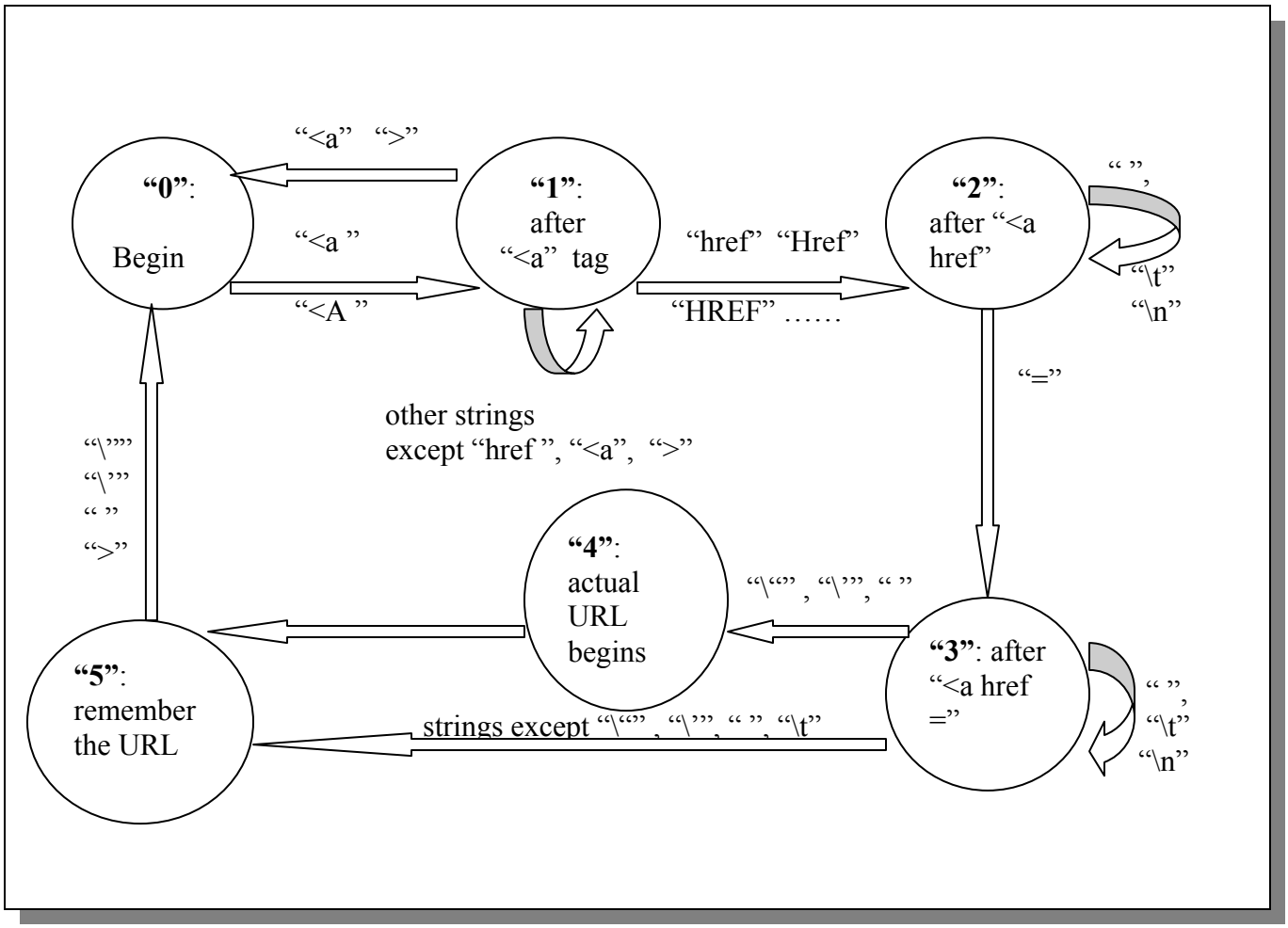

Figure 3. Simplified DFA to get new URLs

The DFA to get a base URL and linkage information is similar to that. The actual DFA in the Parser.java is the one adding all the states and input strings together. Once the DFA gets to state 5, the new URL has been retrieved and it will continue to get the description for this link in the HTML, if it exists. Then the parser will construct a UrlInfo object. UrlInfo.java includes the new URL, the description string, and the URL of the current HTML, which is treated as the parent node for the new URL node. The newly created UrlInfo object is appended to the end of a vector. After the whole HTML file is parsed, the vector holding UrlInfo objects will be returned to the calling object, which is CrawlerThread.

There is a function in the parser class "boolean belongToTargetUrl( )". If the user needs the crawler to crawl through some specific domains, he can write it into the configuration file. When the parser is constructed, it will read those domains' information and put it into targetStrArray. When the parser retrieves a new URL, it can call this method to check if the new URL belongs to the desired domain, and discard the ones outside. If the number of specific domains is very limited, the parser can construct an array to hold the domains and compare the target string one by one with them, otherwise the parser will construct a harsh table and store the domains as the keys in it. Comparing the target string with a hash table reduces the time complexity from O(n) to and ideal of O(1). But the tradeoff is that it will consume more memory.

And there is also a filter function in the parser to get rid of those URLs in form of "mailto: ……" or "javascript: ……". These are not actual URLs, and will waste time if they are put into the new URL buffer.

- Synchronization mechanism:

Since there are simultaneous threads running within each crawler program, there may be some synchronization mechanism to control those threads. For each thread, there are two kinds of database operations involved. They are shown in Figures 4 and 5.

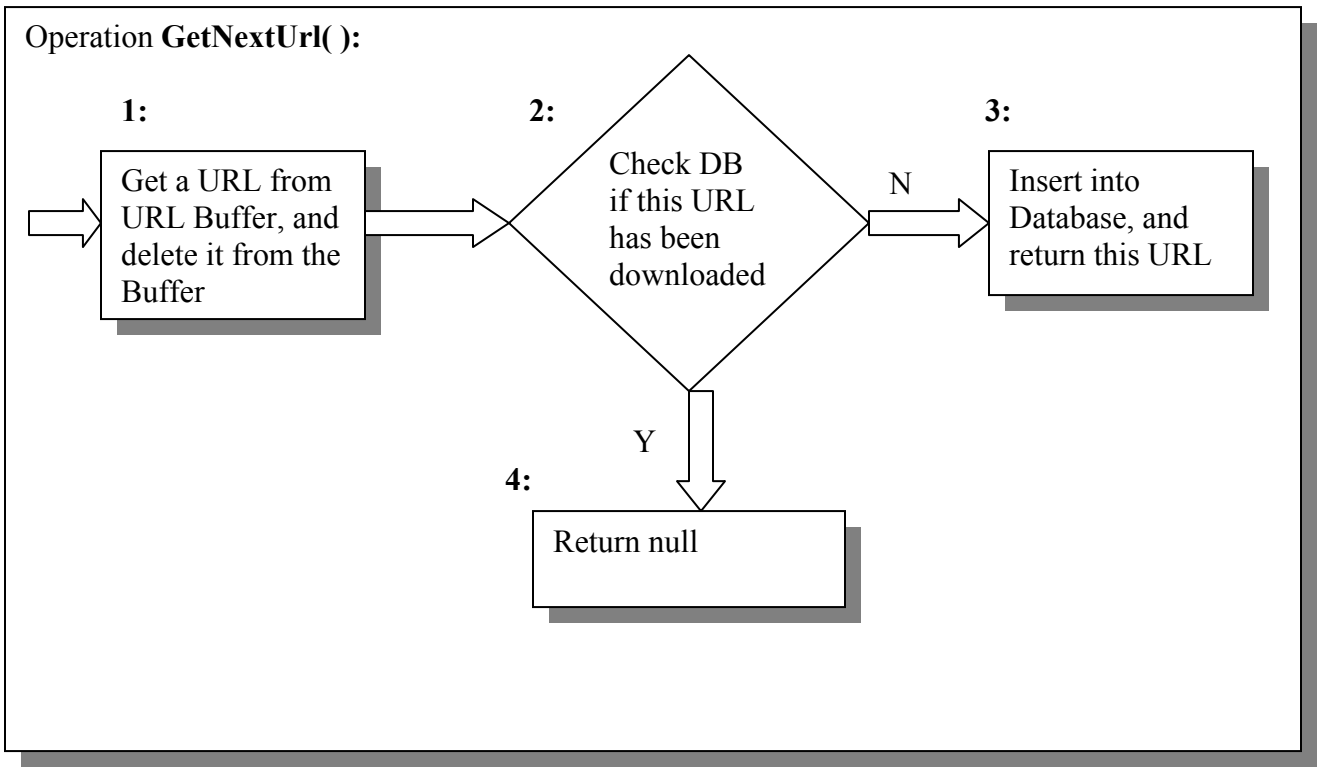

Figure 4. Operation GetNextUrl

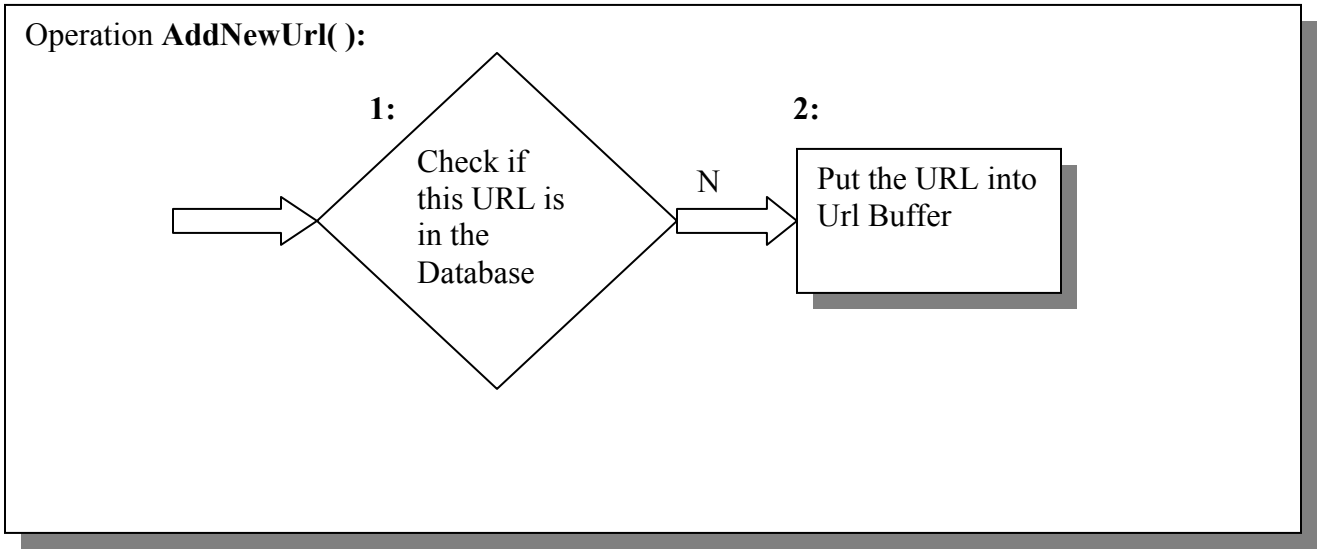

Figure 5. Operation AddNewUrl

If there were no synchronization mechanism, there would be some duplicated URLs in the new URL Buffer. Figure 6 shows an example, in which a redundant URL is inserted into new URL Buffer.

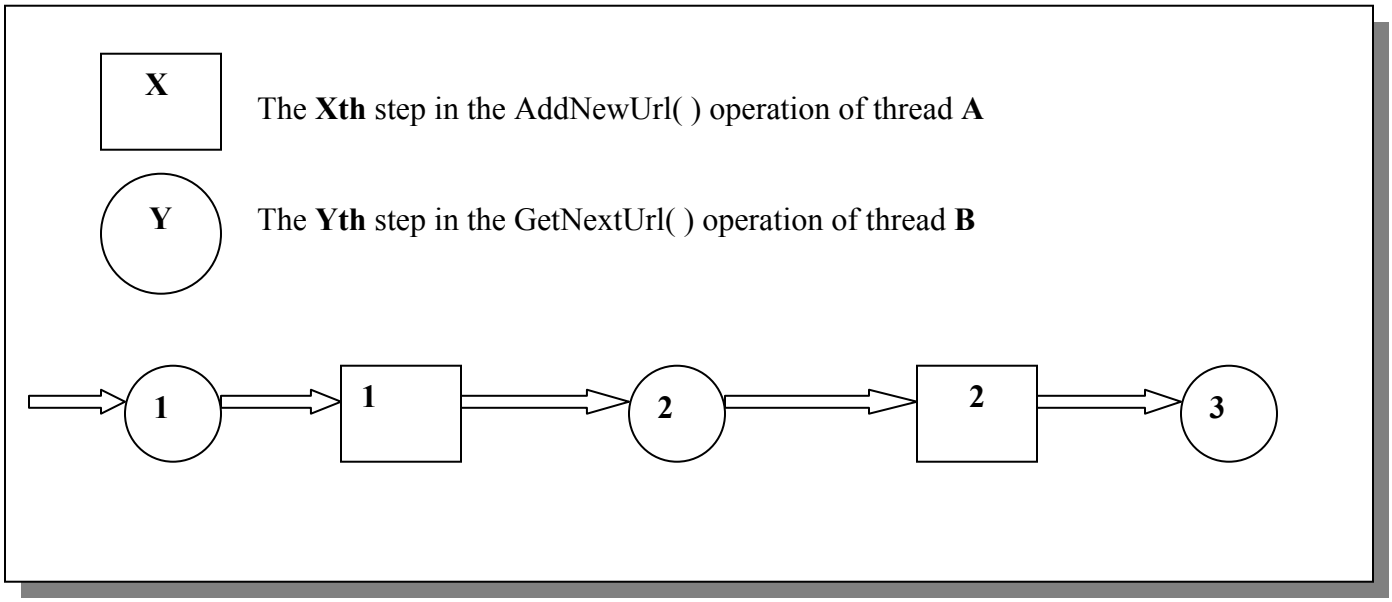

Figure 6. Illustration of interleaved threads

In this case, there is no synchronization between threads, so the operations of threads may be interleaved with each other. In this example, thread B is running the **GetNextUrl( )** method, and thread A is running the **AddNewUrl( )** method. In step 1 of thread B, it gets a URL X from the new URL Buffer. After that, thread A begins to run, and it also wants to insert URL X into the New Url Buffer. After it completes the 1st operation in the method, it finds that there is no URL X in the database. Then, thread B executes the 2nd step of method **AddNewUrl( )**, and it also finds URL X is absent in the

database. In the next operations, threads A and B insert URL X into both the New URL Buffer and the database, which stores all the crawled URLs. After their operations, URL X exists in two places.

If a synchronization mechanism exists, the **AddNewUrl( )** and **GetNextUrl( )** methods can not be simultaneously run by different threads. All the operations within one method will be executed exclusively from those within the other method. With synchronization, the operations in the previous example behave as shown in Figure 7.

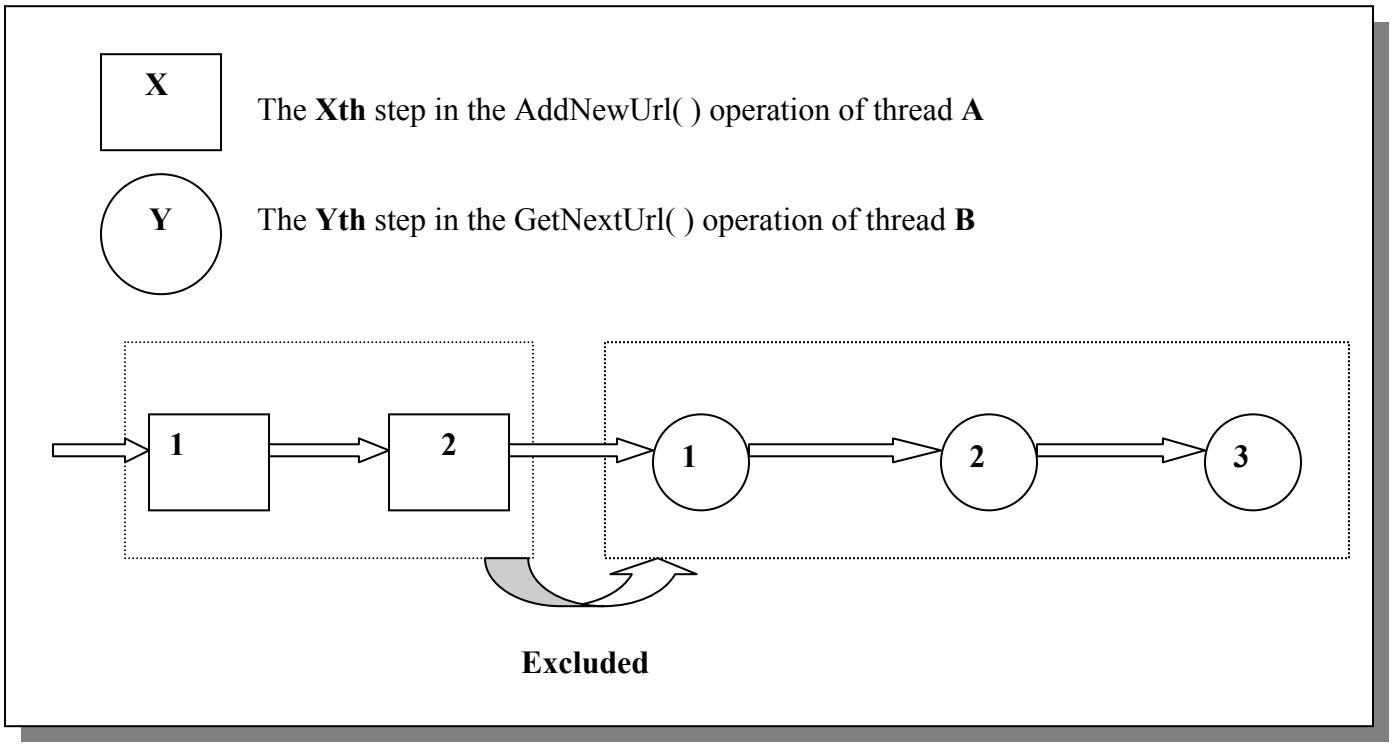

Figure 7. Threads with synchronization

The synchronization mechanism is implemented in the NewUrlBuffer.java. NewUrlBuffer provides three synchronized methods for the threads. "getNextUrl( )" is provided to get the next URL from the URLs Buffer. "addNewUrl( )" inserts a new URL into the buffer. In case of connection failure with a remote connection, "cancelGetNextUrl( )" will be called to delete the bad URL from the database. When those methods are synchronized, the threads are monitored to run them serially. Hence the redundant URLs can be avoided. There is a tradeoff for speed with this method. With synchronization, some originally simultaneous database operations which will not cause redundant URLs are executed serially, so it can reduce the overall crawling speed. Without synchronization, database operations are run simultaneously, but redundant URLs will also slow down the overall performance. In my program, I provide both of these methods.

- Backward Linkage Information:

The WWW can be treated as a network of nodes with directed edges connecting them (Figure 8). The HTML files are nodes, and hyperlinks are directed edges.

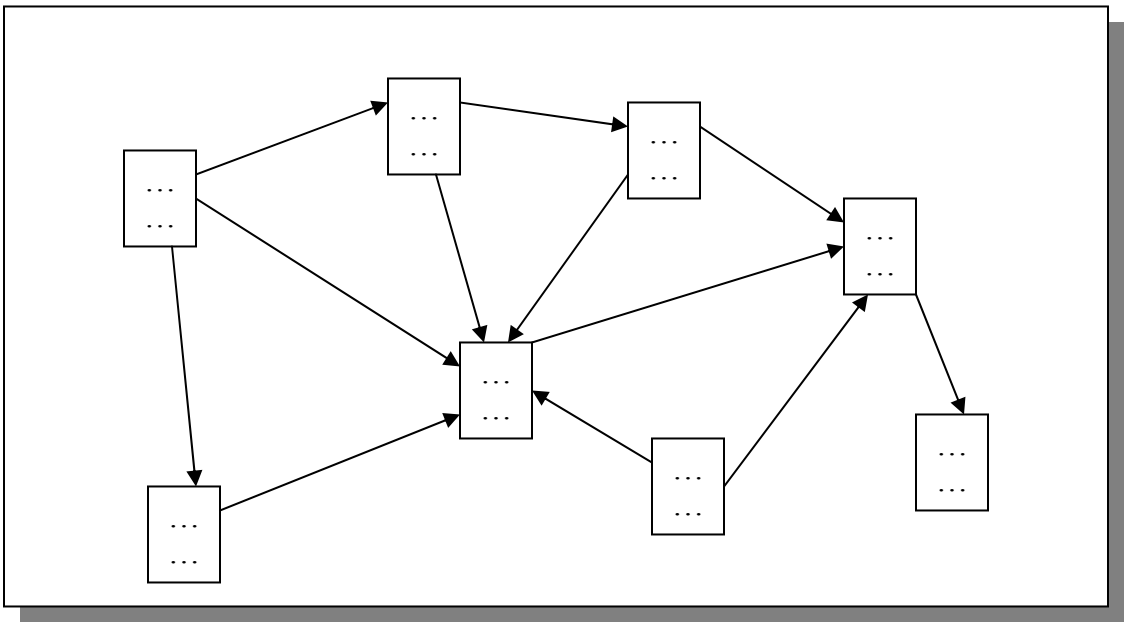

Figure 8. Illustration of WWW as a network

An HTML file may have multiple hyperlinks pointing outside to other HTML files, and at the same time there may be many other files pointing to this file. Backward linkage analysis is a very popular method for Information Retrieval on the Internet, especially for building the Web Searching Engine. The basic idea is to retrieve and analyze the hyperlink description from outside HTML files pointing to the target file, and categorize this file according to those descriptions. If there are twenty HTML files providing a hyperlink to a specific file and they all show "web page of Virginia Tech" on the linkage description, then this file is highly likely to be the web page for Virginia Tech. Those linkages pointing to the target file are called backward links.

An advanced function added to my crawler program is to collect the backward linkage information. If the user chooses to use this function, the crawler needs to run some extra methods and work on another database table. A new class LinkInfoWriter is added, and the NewUrlBuffer class is modified. When a new URL linkage is found in an HTML file, and before it will be inserted into the database, a unique number is created and it represents this newly found URL. The parent URL and the description for the new URL are stored. The next time this URL linkage is found in another HTML file, the description will be retrieved and appended to the end of data associated with it. After the crawler has crawled through enough files, the URLs with the most backward linkage could be found by searching through the database and these URLs should represent popular web sites, because they have so many references from other sites. The

descriptions by other web sites also can be found, and this information is important for the web searching engine building on top of the crawler.

- Word Indexer:

Another advanced function in my crawler is the built-in Indexer module. Indexer is an important module in the web searching engine. The indexer should retrieve all the words in the HTML file and record the URL and position associated with each word. After processing all files, the indexer should build a dictionary with all the words occurring in those files. And each word in the dictionary should be associated with the URLs of the HTML files which contain this word and the positions of word occurrence within that file. The searching engine is built on top of this dictionary produced by the indexer. After users give some key words, the searching engine can look up in the dictionary according to its algorithm and find the most related files with those key words.

In most of the searching engines, the crawler and indexer are built separately. The crawler should be only in charge of downloading web pages and storing them into the hard drive. And the Indexer is built on top of the data crawled by the Crawler. It should only be in charge of retrieving words from HTML files and building the dictionary.

In my crawler, I tried to combine these two modules into one. The reason to combine two modules can be shown in Figure 9:

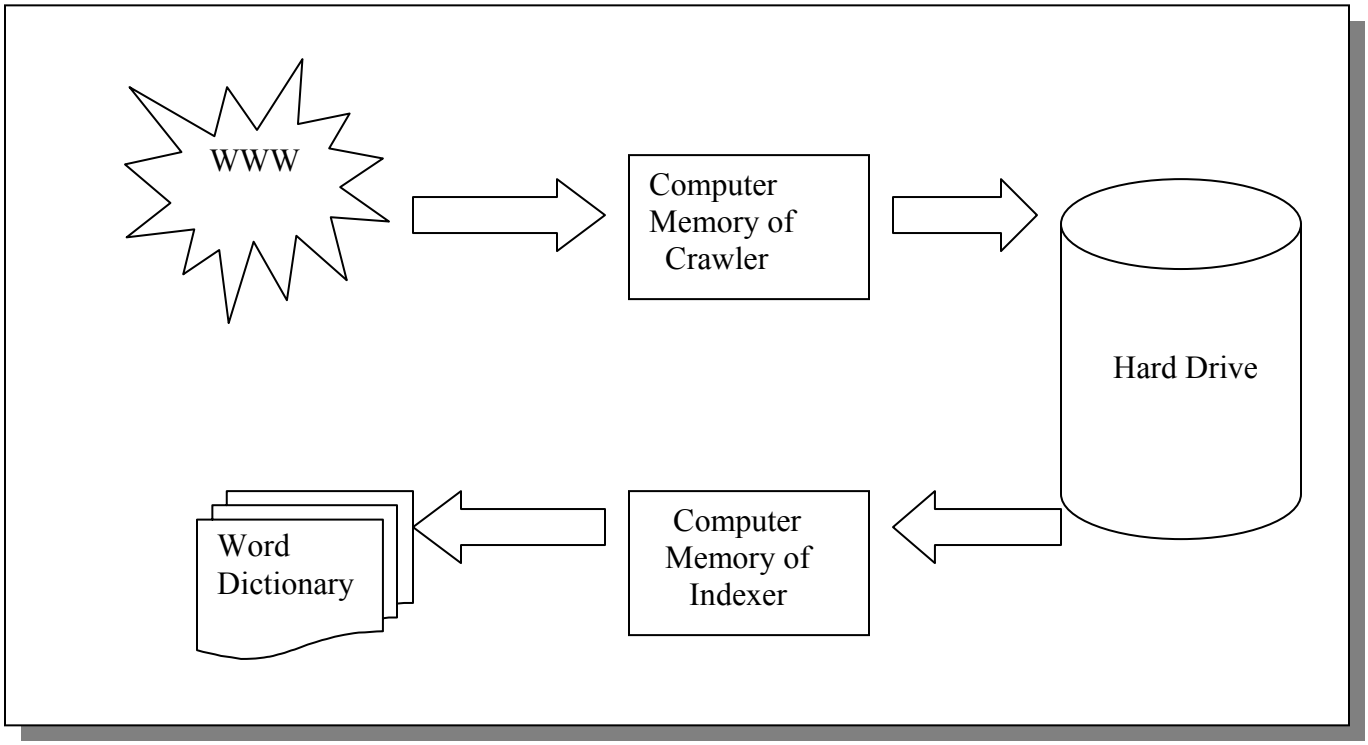

Figure 9. Crawler and indexer

As Figure 9 shows, the crawler needs to crawl on the World Wide Web and store files into the hard drive. But before it can do that, it has to store the files into memory, otherwise it cannot parse these files and retrieve the new URLs. When the Indexer works, it also needs to move the crawled files into the memory from the hard drive, and then create the word dictionary and store it into the hard drive.

The basic idea of my crawler is to create the word dictionary when the HTML files are for the first time moved into memory. After the crawler stores an HTML file into memory, when parsing the file for the new URL linkage and description for this linkage, the crawler will also get rid of anchors, retrieve the pure content, separate the content into words, and build the words dictionary. To support these actions, a new class WordProcessor is created, to write a specific word, its position in the file, and the file name into the database. Two functions, processLine( ) and processWords( ), are added into the CrawlerThread class. These two functions are added to retrieve the pure content from the HTML file and the separated words from the pure content.

Figure 10 shows the basic structure of a word indexer:

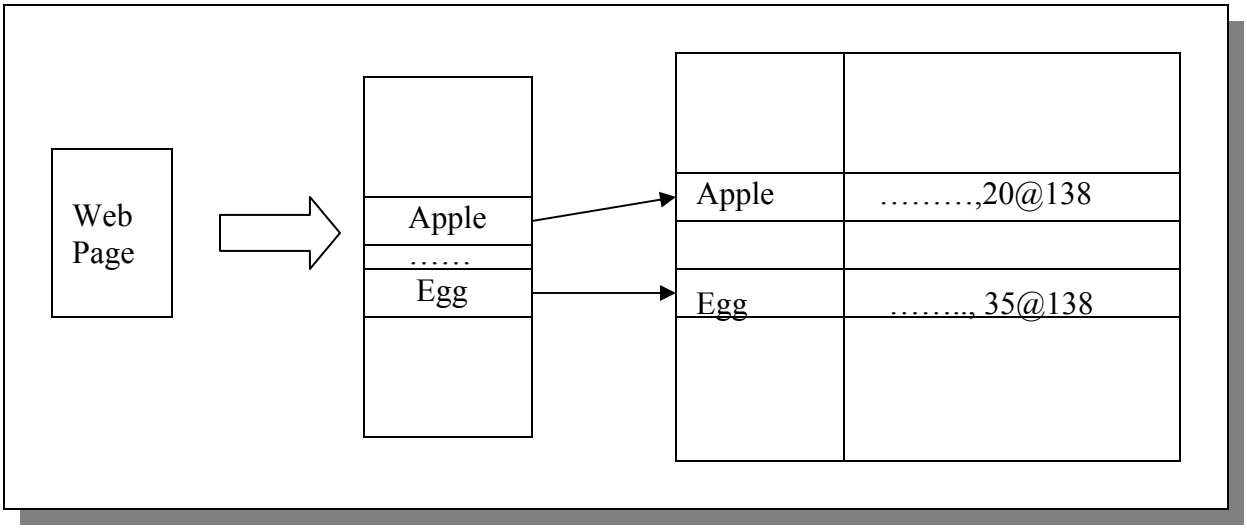

Figure 10. Indexing

After parsing, the content of a file is retrieved. Then the separated words and their location in the file also can be retrieved, and then put into an array. For example, we might find the words 'apple' and 'egg' in a specific file. 'Apple' is the 20th word in this file, and 'egg' is 35th word. The word indexer looks up in the dictionary for those words. 'apple' is found in the word dictionary, so '20@138' is appended to the end of the data field associated with this entry. 'egg' is also found, so '35@138' is appended. '138' is the unique number representing this web page; it is generated when the associated URL is inserted into the database for downloaded URLs.

If the word dictionary is built from web pages that have been downloaded, a module for user query handling can be added on top of it. If a user wants to find a web

page about ‘egg’, the ‘egg’ entry will be located in the word dictionary. And all the web pages that contain ‘egg’ can be found. The query module has not been built in this project. If it were created, this project could work as a mini web search engine.

- Performance Tuning:

An important feature of the crawler is its efficiency. It needs not only to retrieve new URLs and other information correctly, but also to finish the operations as soon as possible. After the crawler began to run, a good deal of my time went into performance tuning. Eight versions were made to measure the performance changes as the different parameters change. In each version there are some modifications in parser design, database structure, and framework – relative to the previous version.

Performance tuning is a time-consuming endeavor, since there are many factors affecting the performance of the crawler. The major factors inside the crawler are number of crawlers and crawler threads, database design, parser performance, and relations for crawler threads. The major environmental factors are traffic on the Local Area Network (LAN), traffic on the remote server, and the system configuration. Sometimes those environmental factors are the dominating factors regarding crawler performance. For example, when I made the crawler crawl on a specific domain www.chinaren.com, a popular Chinese website whose server is located in China, the crawler’s performance difference varies significantly as a function of its running time. When I run the crawler at around 8:00 am at Virginia Tech, the performance is poor (about 8 MB per minute). The Chinaren.com’s web server on the other side of the earth is kept busy at that time, because it is around 8:00 pm in China and many users are accessing the web server. In this case, the traffic on the remote server is the dominant factor. When I run the crawler at around 4:00 pm, it achieves better performance (about 15MB per minute), because only the most enthusiastic users are still online at that time in China and its web server is not very busy. But when I run it at around 6:00pm, it always achieves the best performance (about 20MB per minute), because at that time the LAN in Virginia Tech is not busy either. As you can see, without changing the crawler program, the performance difference is significant. To tune the performance, I use the ‘single factor’ approach, keeping other factors unchanged and measuring the performance changes with only a single factor changing. However some environmental factors cannot be controlled, like the traffic on the remote server and local area network. Every time one measures the performance with regard to a specific interior factor, the crawler should be run for long enough to cancel the influence of fluctuating environmental factors. That’s why performance tuning is so time-consuming.

The major interior factors of the crawler include database structure, parser design, and the framework of crawler threads. There are several major modifications on each factor. Figure 11 shows the modifications to the database design:

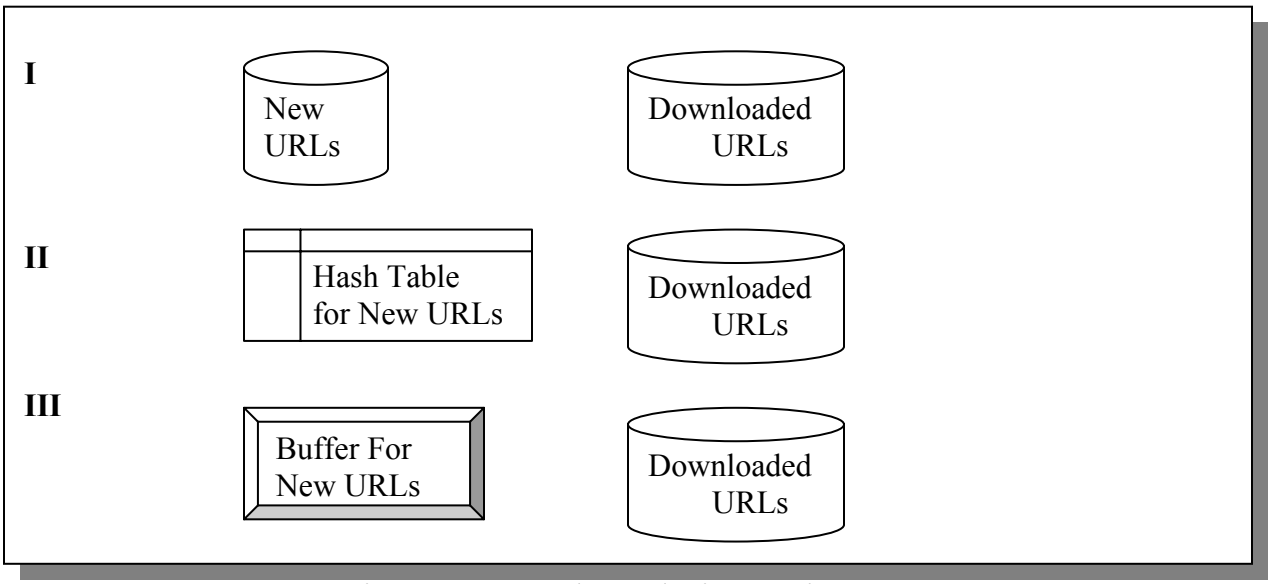

Figure 11. Database design options

In the first version, there are two databases; one is used to store new URLs found in the HTML files, while the other one stores the URLs that have been downloaded by the crawler. The URL string is the primary key for both databases, so there are no duplicated URLs in each database. Inserting a URL string which existed in the database will cause a "duplicated key" error. In this case more time will be spent on handling the error than checking the existence of a URL before actually inserting the URL. Hence after a new URL is found and before it is inserted into the new URL database, it will be checked twice. The first is to check if there is such an entry in the downloaded database, while the second is to check if it exists in the new URL database. After checking, if it doesn't exist in either one, the new URL will be inserted into the new URL database.

In the next version, the database for new URL strings is replaced by a hash table. A hash table works as a database, but it can store the information in memory. Using the hash table instead of a database can save some time on searching and inserting. In my experiment, a HTML file can provide about ten new URL linkages on average. In the database the entries of new URLs are ten times more than those of downloaded URLs. If there are too much data stored in the hash table, it will finally swap out some data in memory onto the hard drive. After a substantial number of web pages are retrieved, using a hash table will lower the speed, since a lot of time is spent on swapping data in and out. Hence in this version, a threshold of hash table size is added. When the number of entries in the hash table exceeds a specific number, the following new URLs found will be discarded. In this version, for each new URL there are also two checks, on the downloaded URL database and new URL hash table. However, since a hash table is used, the performance is better than that of the previous version.

In the current version, the hash table is discarded and a buffer is used. The new URLs are simply inserted into the buffer, and no checking is performed. In the previous versions, a more restricted rule is implemented, so there are no duplicated URLs between the databases as well as within each database. Now such a rule doesn't exist, so there may be duplicated URLs in that buffer and some URLs in the buffer may exist in the database for downloaded URLs. We can save some time without performing any checking on the databases for each new URL. But when a thread gets a URL from the buffer, it is possible that the same URL has already been in the database for downloaded URLs. In this case, the duplicated URL obtained by the thread is useless and the time spent on inserting and retrieving this URL will be wasted. That's the tradeoff for this approach. However, after performance measurement, this method is shown to achieve the best performance relative to the previous ones, although it doesn't look elegant.

There are also some modifications on the parser's structure. To retrieve the desired information, we need some functions to perform checking according to regular expressions, however Java doesn't provide such functions. The underlying mechanism for those functions is the Deterministic Finite state Automata (DFA), so the parser is designed as a DFA. A DFA has a finite number of states and the state changing can be completely determined by its current state and input. There are two versions of the parser. In one version, inputs of the DFA are characters and string functions in Java are not used. The HTML file is treated as a collection of characters. In the other version, inputs are strings, the string functions in Java are used and the HTML file is treated as a collection of strings. In the first version the DFA has many more states than the second one, since there is a state between every two characters. To check the next state with an input character, it needs to check through a larger table than the second version parser checks through. Although the second parser uses the time-consuming string manipulation functions, according to the performance comparison it has the better performance.

The framework of the crawler thread is another factor affecting the performance. Figure 12 shows the operations of the thread in the earliest phase.

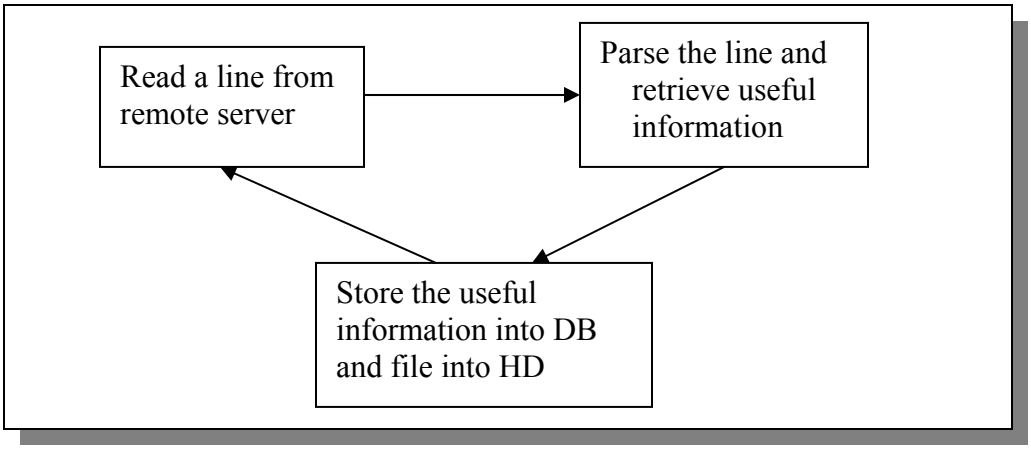

Figure 12. Crawler thread activity, version 1

In this version, there is only one kind of thread, which is in charge of everything. To save memory, the thread doesn't store the whole HTML file into it, but reads a line, processes it, and then reads the next line. All the threads are identical and running simultaneously.

Figure 13 shows the framework in the second version:

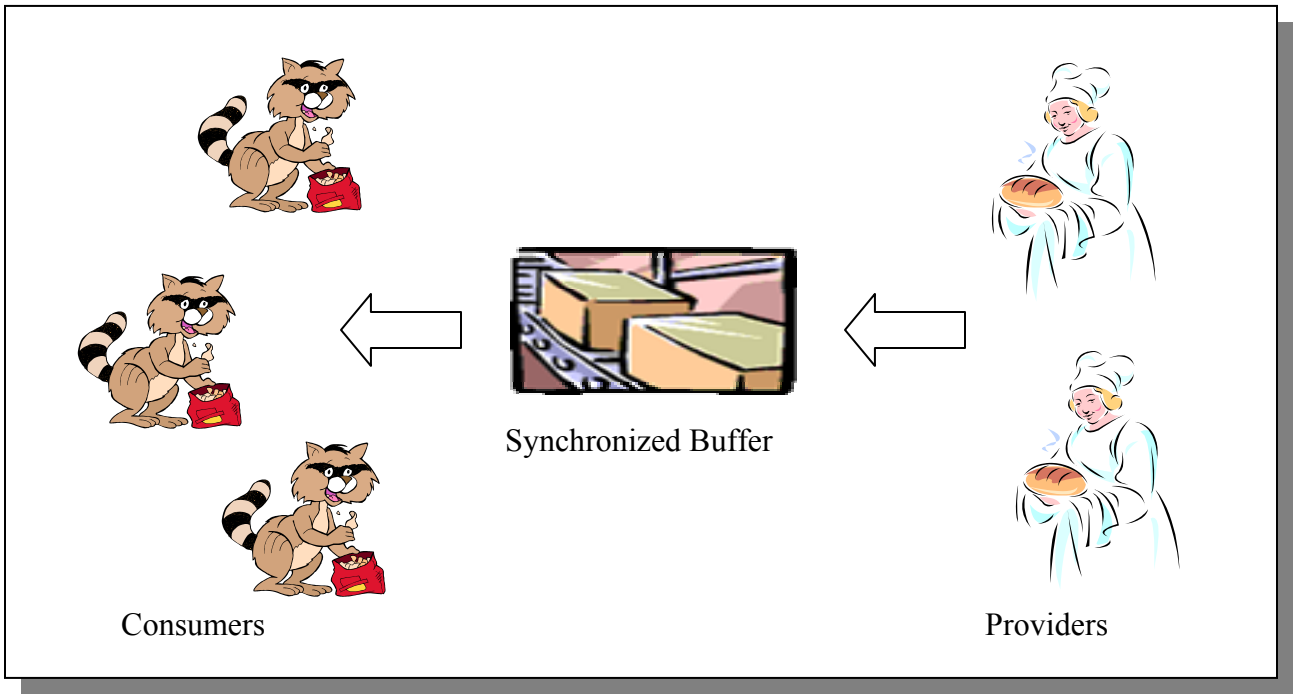

Figure 13. Synchronization in version 2

There are two kinds of threads in this version. Some threads serve as providers. What they do is to retrieve the web page from the server, store the HTML file in the memory, and put the reference of the memory block into the buffer, which consumers can access. Other threads work as consumers. They get the HTML file from the buffer, parse the content, insert useful information into the database and store the HTML file on the hard drive. All of those threads are running simultaneously. The synchronization mechanism in the buffer can prevent the data from being corrupted, so the HTML files are actually inserted into and taken out of the buffer linearly by the simultaneous threads. The performance of this version is better than the first one, probably because it retrieves the whole HTML file and stores in memory, instead of processing it line by line. But the shortcoming of this structure is that both consumers and providers are kept synchronized by the same buffer. The buffer becomes the bottleneck of threads. If there are a substantial number of threads running, a lot of time will be used for synchronization.

To reduce the time spending on synchronization, the third version is created. In this version, the centralized buffer is discarded, so the synchronization mechanism controlling all the threads doesn't exist. This version is shown in Figure 14:

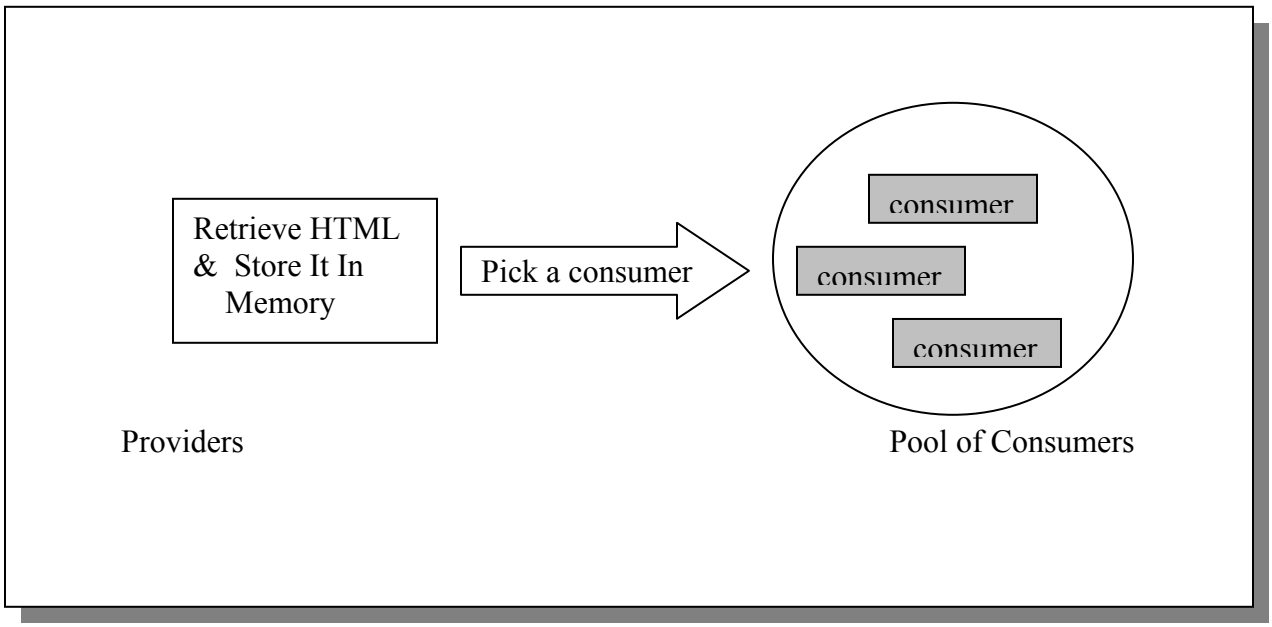

Figure 14. Crawler thread activity, version 3

There are still two kinds of threads, consumers and providers. The difference between this version and previous one is that after the provider retrieves a file and stores it in the memory, it will pick a consumer thread from the pool and pass the reference of that memory block directly to the consumer. In this case the situation that multiple consumers and providers compete for the access to the buffer can be avoided. The performance is improved, but not significantly.

Finally this consumer and provider structure is discarded. In the current version, there is only one kind of thread. Figure 15 shows the thread structure. Now all the tasks are performed by identical threads. As Figure 15 shows, the thread first retrieves the whole file from web server and stores it in the memory. After it calls a method to process the file from memory, the thread writes it on the hard drive. There are no direct communications between threads any more. After measurement, this structure achieves the best performance.

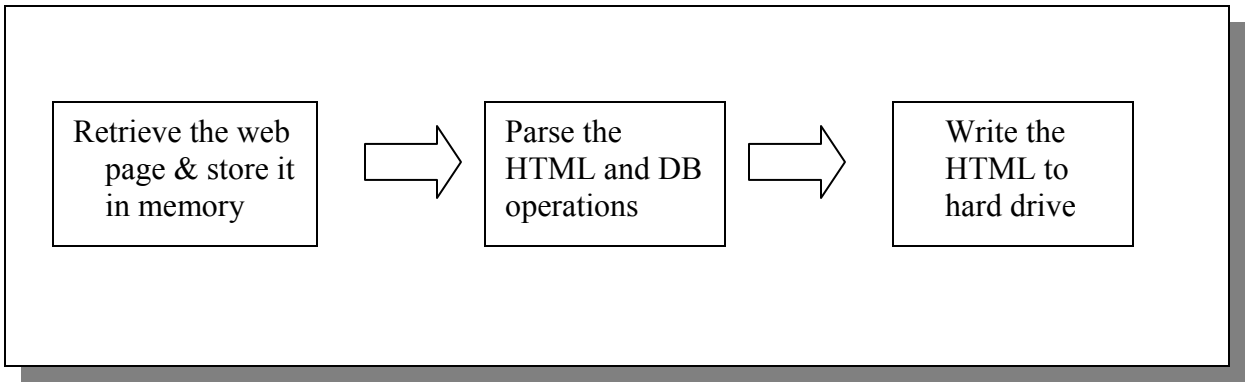

Figure 15. Crawler thread activity, version 4

3. Source Code Explanation:

In this section, a brief explanation for major functions of each class is provided.

- DBDriver.java

  DBDriver class sets up the database connection to the MySql DBMS. There are noofficial Java driver programs for MySql DBMS. From the World Wide Web, I found a driver org.gjt.mm.mysql.Driver, written by a Ph.D. in Purdue University. After installing it into my computer, it works well with MySql DBMS. So my crawler connects to the database through his driver.

  The constructor DBDriver( ) builds an instance of driver org.gjt.mm.mysql.Driver, sets up the connection to the database, and gives the user name and password to the DBMS for authorization.

  The method "public Statement generateStatement( )" is used to generate a Statement object, which is used for actual database operations from crawler threads.

- ConfigManager.java

  ConfigManager class reads the configuration file "conf.crawler", parses the content, makes a judgment if the configuration file is correct, and provides information for other classes according to the configuration.

- conf.crawler

  It is a pure text file, used for configuration. The format of it is "Key = Value ". If a key associates with more than one value, the values are separated by semicolon. Now there are seven fields in this file. They indicate how many threads are expected to run within each crawler, if the crawl is limited to a specific domain, the desired domains for the crawler, etc. There are detailed descriptions in this file.

- Crawler1.java

  It contains the main( ) function and it is the file to run the crawler. The tasks for this class are simple. It builds a ConfigManager object to read the configuration file, builds the DBDriver object to connect to a database, and creates some crawler threads according to the configuration file. It also monitors the thread activities, counting the number of active threads and number of entries in the buffer for new URLs.

- CrawlerThread.java

  It is the source code for crawler threads. It is a subclass for Thread class. There are two major functions "grab_webPage( )" and "parseHtml( )". After an instance of CrawlerThread class is created and its start( ) method is called, the crawler thread is activated and begins to crawl on the Web. First it calls grab_webPage( ) to retrieve a web page. After it gets a URL from the new URL buffer, the thread will contact the web

server for the header fields of the web page. Header fields indicate the type of this file, the total length of it, and other information. According to the header fields, the thread can make the decision whether or not to download this web page. If the file is desired, the thread will create a dynamic array, to store the file content. For pure text files, the thread downloads them through a BufferedReader, which reads the file character by character. Otherwise, the thread reads them through a BufferedInputStream, which reads the file byte by byte. Then "parseHtml( )" is called, and the reference of the dynamic array is passed to it. Each thread contains an instance of the Parser class. The parseHtml method calls methods in the Parser class to retrieve the useful information in the file. And in the CrawlerThread class there are also two methods called processLine( ) and processWords( ), which help parse the file. There are detailed comments in the source code about those methods.

- Parser.java

  Parser.java is the source code for the parser. The major functions in this class are "parseIt( )" and "parseItWithInfo( )". The "parseIt( )" function retrieves new URLs in the file and insert the new URL strings into a dynamic array. The "parseItWithInfo( )" function can retrieve more information from the file. It inserts an instance of the UrlInfo class, which contains information about the new URL and linkage information, into the dynamic array.

- ThreadMonitor.java

  It is the class to monitor the activities of threads. When the threads are running, it provides some feedback for the user about how many threads are currently active and how many entries the new URL buffer has. It has a synchronization mechanism to prevent its data from being corrupted.

- UrlInfo.java

  This class has three fields: new URL string, linkage information string, and parent URL string. In the Parser class, the instance of UrlInfo class contains all the information for each new URL and is inserted into the dynamic array.

- UrlInfoNum.java

  This is the subclass of UrlInfo. It inherits all the data fields in the UrlInfo class and has one more data field, file number, which uniquely represents a URL. The new URL buffer is a dynamic array that has a synchronization mechanism and stores instances of the UrlInfoNum class. After a thread gets an object from the new URL buffer, it can get the new URL string, its linkage information, its parent URL and a unique number, which is associated with the new URL string.

- LinkInfoWriter.java

This class is used for the backward linkage information collector. It has two major functions "createNewInfoFile(UrlInfoNum u)" and "createNewInfoFile(UrlInfoNum u)". If a URL is met the first time, createNewInfoFile( ) is called, and a file associated with this URL is created. Because an instance of UrlInfoNum class is passed as its parameter, the createNewInfoFile( ) method will write the information contained in the instance into the file. If a URL has been retrieved and is met again in the crawling process, createNewInfoFile( ) will be called. This method locates the description file associated with the URL and appends the information contained in the UrlInfoNum instance to the end of that file. In this way, all the backward linkage information for a specific URL will be stored in the file associated with that URL. These two methods write data to the file in the same format, which is shown in detail in the source code. Thus, all the backward linkage information can be easily retrieved from those files.

- WordProcessor

WordProcess class is used to build a word dictionary. It maintains a connection to the database. It has only one method "addToDict(String word,long fileNum,int position)". Given a specific word, the number associated with the URL, and the position of the word's occurrence, this method will ask the database to locate the word in the dictionary and append a "position@fileNum" string to the end of data associated with the word.

4. Future Work:

When the Backward Linkage Information Collector and the built-in Indexer work, the crawling speed is slowed down significantly. When deploying one crawler on my computer, the average crawling speed is about 10 MB per minute without these advanced functions. But if I choose to run these functions on top of the crawler, the average speed is about 500KB per minute, which cannot be accepted in a crawler program. After careful code inspection and performance tuning, the crawling speed still cannot be increased.

I think that happens probably because of the poor running efficiency for the Java program. As I mentioned in the first section, a simple crawler program is an I/O-intense program. The performance of programs written in Java is acceptable, and its platform independent feature outweighs its slower running speed. However, after I implement the basic functions and add more advanced functions, the original I/O-intense crawler becomes a CPU-intense crawler. The new crawler needs to retrieve more information from the files, which involves more computation and more database operations, so CPU load is significantly increased. Since Java is an interpreted language, it needs much more

work for the same operations compared with traditional languages. Under these circumstances, Java is not the correct programming language. The whole program should be rewritten with C++ or other non-interpreted language, if we want to combine those Backward Linkage Information Collector and built-in Indexer modules.

For the algorithm to crawl the WWW, I implement both the depth first and the breadth first search in my crawler. The user can switch between these two searching algorithms. There are a lot of papers on the web about crawling algorithms. Some of them mention a more intelligent and user adaptive algorithm. But that algorithm involves much more computation to decide the next URL to be crawled. Developing a more intelligent crawling algorithm can also be a future research direction.

REFERENCES

[1] Jenny Edwards, Kevin McCurley, John Tomlin. An Adaptive Model for Optimizing Performance of an Incremental Web Crawler, WWW10, May 2001, Hong Kong
[2] Marc Najork, Janet L. Wiener, Breadth-First Search Crawling Yields High-Quality Pages, WWW10, May 1-5, 2001, Hong Kong
[3] Paraic Sheridan, Martin Wechsler, Peter Schauble. Cross-Language Speech Retrieval: Establishing a Baseline Performance, SIGIR 97 Philadelphia PA, USA
[4] Junghoo Cho, Narayanan Shivakumar, Hector Garcia-Molina. Finding replicated web collections, MOD 2000, Dallas, TX USA
[5] Charu C. Aggarwal, Fatima Al-Garawi, Philip S. Yu. Intelligent Crawling on the World Wide web with Arbitrary Predicates, WWW10 May, 2001, Hong Kong.
[6] Sougata Mukherjea, Organizing Topic-Specific Web Information,
[7] Michael Chau, Daniel Zeng, Hsinchun Chen. Personalized Spiders for Web Search and Analysis, JCDL '01, June 24-28, 2001, Roanoke, Virginia, USA
[8] Arvind Arasu, Junghoo Cho, Hector Garcia-Molina, Andreas Paepcke, and Sriram Raghavan. Searching the Web, ACM Transactions on Internet Technology, Vol. 1, No. 1, August 2001, Pages 2-43
[9] Filippo Menczer, Gautam Pant, Padmini Srinivasan, Miguel E. Ruiz. Evaluating Topic-Driven Web Crawlers, SIGIR '01, September 9-12, 2001, New Orleans, Louisiana, USA
[10]Charu C. Aggarwal, Fatima Al-Garawi and Philip S. Yu. On the Design of a Learning Crawler for Topical Resource Discovery, Vol. 19, No. 3, July 2001