MCMTCrawler: a Multi-Computer and Multi-Thread Vertical Crawler

Ziyun Deng, Lei Chen, Tingqin He and Tao Meng

***Abstract*—To optimize the structures of the open source crawlers, improve the performances of the standalone crawlers, we design a new Multi-Computer and Multi-Thread vertical Crawler, called MCMTCrawler. MCMTCrawler can complete the special crawling task on a large business website within a few hours. MCMTCrawler uses Berkeley DB to persist the waiting Uniform Resource Locator (URL) queue and the downloaded URL queue. MD5 algorithm is applied to map a URL to a 32-length string. MCMTCrawler employs the Producer-Consumer model to assign and process the URLs. Based on the design ideas of Aspect-Oriented Programming (AOP) and Dependency Injection (DI) of Spring, the scheduler and the downloader of MCMTCrawler are designed separately for speeding up the crawler. According to the experimental results, when using three downloaded servers, the speed of MCMTCrawler is five times as much as that of the single-computer and single-process crawler, and three times of the single-computer and multi-thread crawler called Crawler4j. Furthermore, for handling the task of crawling 600,000 web pages, MCMTCrawler takes only 6.83 hours.**

***Index Terms*— MCMTCrawler, Multi-Computer and Multi-Thread, Vertical Crawler, Design idea**

I. INTRODUCTION

IN the era of big data, it is a commonly mean to use big data for analyzing and solving realistic problems in the business applications [1]. To apply the technology of big data, we must have enough data in the first. The amount of data is greater, the probability and accuracy are higher for analyzing and solving the problems [2]. There is a broad amount of data on the Internet. It is obviously unrealistic to centralized store all of the data. However, for the specific domains, we can use the crawler to download the focused data from the related web pages, and further to extract the value of these data.

Therefore, our team designs a crawler to download the web pages from several large-scale commercial websites, and extract the commodity information for building the special database. We further analyze the price trend of these goods, thus provide the purchasing suggestions for the buyers, and offer the price recommendations for the sellers. The research ideas of our team and the focus of this paper are shown in Fig.1.

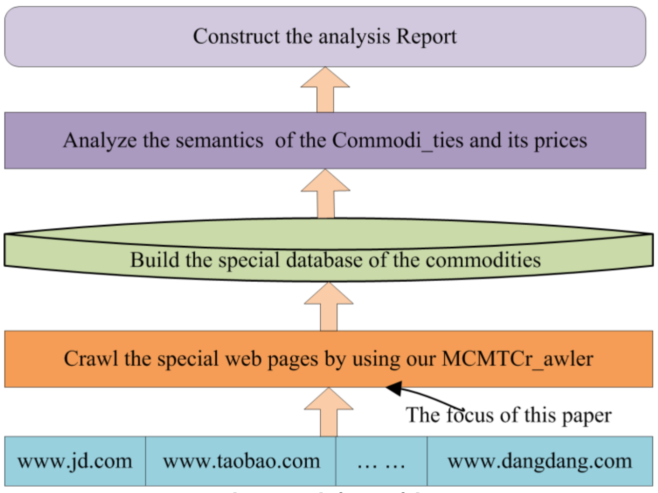
Construct the analysis Report
Analyze the semantics of the Commodi_ties and its prices
Build the special database of the commodities
Crawl the special web pages by using our MCMTCr_awler
The focus of this paper
www.jd.com
www.taobao.com
… …
www.dangdang.com

Fig.1. The research focus of this paper

Generally, a large-scale commercial website has a huge number of web pages, which can reach the level of several hundreds of thousands [3], even the level of several millions. To crawl on the web pages in this website, we must have a high-speed crawler. Let us take a simple example. If a large-scale commercial website has 500,000 web pages, and a crawler takes one second to download one web page, then the crawler crawls on only 86,400 web pages in 24 hours. The speed of crawling is obviously inefficient. Therefore, the single-process structure of the crawler is no longer satisfied with current need, so it is very important to design a high-speed crawler. In this paper, we don't care about the related analysis of the web content and the commodity price. We only focus on the design of a Multi-Computer and Multi-Thread crawler called MCMTCrawler. MCMTCrawler can quickly download the web contents from any large-scale commercial website. We hope the MCMTCrawler can crawl hundreds of thousands of web pages every day.

The remainder of this paper is organized as follows. We describe the background and related work in section II. In section III, we present our new crawler called MCMTCrawler. We show the experimental evaluation in section IV. Finally, we draw our conclusions in section V.

Manuscript received January 21, 2018. This work was supported by the National Key Technology Support Program of China (No. 2012BAH09B02); the Natural Science Foundation of Hunan Province (Nos. 2017JJ5064); the Social Science Foundation of Hunan Province (No. 16ZDA07).

Ziyun Deng is with the College of Economics and Trade, Changsha Commerce & Tourism College, Changsha 410116, China (corresponding author to provide phone: +86-138-7492-1889; e-mail:dengziyun@126.com).

Lei Chen is with School of Information and Electrical Engineering, Hunan University of Science and Technology, Xiangtan 411201, China (e-mail: chenlei_hnust@126.com).

Tingqin He is with National Supercomputing Center in Changsha, Hunan University, Changsha 410082, China (e-mail: hetingqin@hnu.edu.cn).

Tao Meng is with National Supercomputing Center in Changsha, Hunan University, Changsha 410082, China (e-mail: mengtao@hnu.edu.cn).

II. Background and Related Work

In the past several years, a lot of related research works and achievements can be summarized into three aspects.

A. Optimization of task queue

Based on the previous literature researches [4-7] and some open source crawlers [8-10], a crawler should generally have at least 4 components including downloader, waiting Uniformed Resource Locator (URL) queue, downloaded URL queue, and scheduler. The downloader refers to crawl the web pages according to the waiting URL queue. The waiting URL queue is in charge of storing the list of URLs to crawl. The downloaded URL queue refers to store the list of URLs that have been downloaded by the crawler. The scheduler sets up a series of crawling rules, and drives the crawler to work. In the traditional web crawlers [4-17], First Input First Output (FIFO) algorithm is widely used to transfer the URLs from the waiting URL queue to the downloaded URL queue. However, FIFO algorithm usually causes some web pages couldn't be crawled after a long time. The main cause of this phenomenon is to use the depth search algorithm for finding the web pages [15, 18]. Based on the above reason, some researchers apply some better search algorithms and in-memory queues to optimize the task queue for improving the crawler speed. The crawlers [6, 11] use the broad search algorithm. The works [19, 20] employ the scheduling algorithm based on the URL classification to further improve the crawler speed. Otherwise, some crawlers [5, 11, and 17] use the table of the database as queue structure to store the URL list, which is very good for data persistence. But, these crawlers usually have a low speed. For that, the newly developed crawlers [8, 14, and 19] use the memory queue structure or the simple Key-Value database under the enough memory condition.

B. Optimization of persistent storage

To improve the speed, three optimized ways of persistent storage are used for the waiting URL queue and the downloaded URL queue. They are the shared memory queue, filter and the memory database. In the first way, HashSet is typical. If we use the shared memory queue, the memory will easily overflow. Moreover, once the program is interrupted, the data in two queues will be automatically cleared without persistence processing. In the second way [20], BloomFilter is typical. We need to set up three different parameters (m, k, n) for getting the best result for BloomFlter. Because these parameters are dynamical, it is very difficult to get the ideal values for the crawlers, and it is still possible to produce misjudgment or memory overflow. Therefore, the way using the shared memory queue and the way using the filter are both more efficient than the way using the memory database. The way using the memory database, such as Berkeley DB, is the compromise solution and more stable than other ways.

C. Optimization of structure design

To further improve the crawler speed, the structural design of the crawler is optimized. For example, multi-threaded and distributed technology is used in the structural design of a web crawler. Currently, many crawlers [4-17] use the structural design, such as Crawler4j[8], Nutch[9], Web collector[10]. All of them support multi-threaded downloader or distributed deployment. In theory, if the network bandwidth is enough, then the speed of the crawl for downloading web pages is faster through the more threads and more computers participate in working. However, when a multiple-threaded or distributed crawler runs, the downloader and scheduler get the URLs at the same time from the waiting URL queue. This leads to the problem of reading some URLs repeatedly [21]. The waiting URL queue hasn't Atomicity/ Consistency/ Isolation/ Durability (ACID) features like the database system. Furthermore, the existing crawlers put the downloader and the scheduler working in the same thread, the downloader takes a large amount of work time in the thread, and the scheduler is also waiting.

To solve above three problems including, "the downloader and scheduler isn't separated in one thread", "reading some URLs repeatedly in a multi-threaded crawler", and "memory overflow", this paper designs a new crawler, called MCMTCrawler.

III. MCMTCrawler

A. Berkeley DB persistent

Berkeley DB is used for persistent processing the waiting URL queue and the downloaded URL queue, as showed in Fig.2. In the waiting URL queue, the key records URL. Considering the waiting URL queue only needs to record the URLs, the value is a static byte, so that the whole waiting URL queue only needs 1 byte to record the value. If we simply store the URLs of the Key-Value data structure, then it's difficult to handle the different URL lengths, the consumption of storage space is becoming large, and the search speed will become slow. Therefore, MD5 [22] calculation can be used to map a URL to a 32-length string as the URL's unique identifier. The URLs have the same length after the MD5 calculation is executed. The calculated MD5 result is used as the key index for searching the URL queue with fast downloading speed. If a match item is found, then it means the URL has been downloaded.

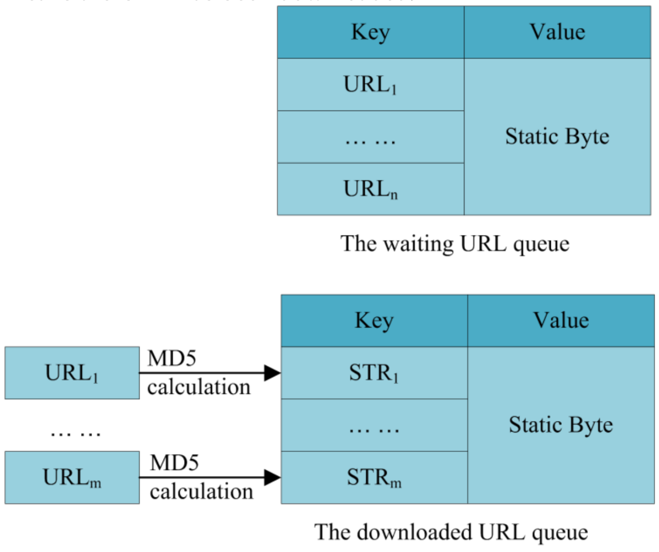

Fig.2. The data structures of the waiting URL queue and the downloaded URL queue

MD5 calculation mainly consists of three steps. Firstly, the calculation converts a URL to a 16-byte long array using MessageDigest class of Java Development Kit (JDK). Then, the calculation traverses the element of the byte array, and gets the absolute value. For each absolute value, the calculation parses the value into a string with 16-hexadecimal format. If the length of the string with 16-hexadecimal format is less than 2, then the calculation fills in zero in front of the string. Finally, the calculation merges these strings, and gets a 32-length string. So, in fact, the capacity of this 32-length string is $16^{16} = 2^{64} = 16$EB, which is huge and enough for the vertical crawlers. For the consumption of the storage space, before the MD5 calculation, the length of the estimated URL is about 120-150 strings, which are 4-5 times storage space required than the length of the URL after the MD5 calculation. Assuming a large commercial website has more than 5 million URLs. The required storage space for MD5 calculations is the following value,

$$5{,}000{,}000 \times 32 \times 2 = 320 \text{ M Byte} \qquad (1)$$

If we skip the MD5 calculation, then we need 1.28G ~1.6G storage space at least. In addition, by using Berkeley DB, the memory will never be overflowed if we ensure enough hard drive space. Thus, because of uniform length and shorter strings as key, the design helps to search for Berkeley DB.

B. "Producer - Consumer" model

The Producer-Consumer model usually starts a thread specifically to accomplish the functions of "Producer". Firstly, the thread judges whether the buffer is full. If the buffer is not full, the thread gets the first URL from the waiting URL queue. The thread makes accessibility test after the thread uses url comparing with the downloaded URL queue. If the test is successful, the thread pushes url into the URL queues for the scheduling threads. So the producer generates the URLs. When thread start, it generates k static URL queues for the scheduling threads according to the number of dispatch threads injected by Spring at the initial startup. Which URL queue for the scheduling thread should be url pushed into? The algorithm is showed in Table I.

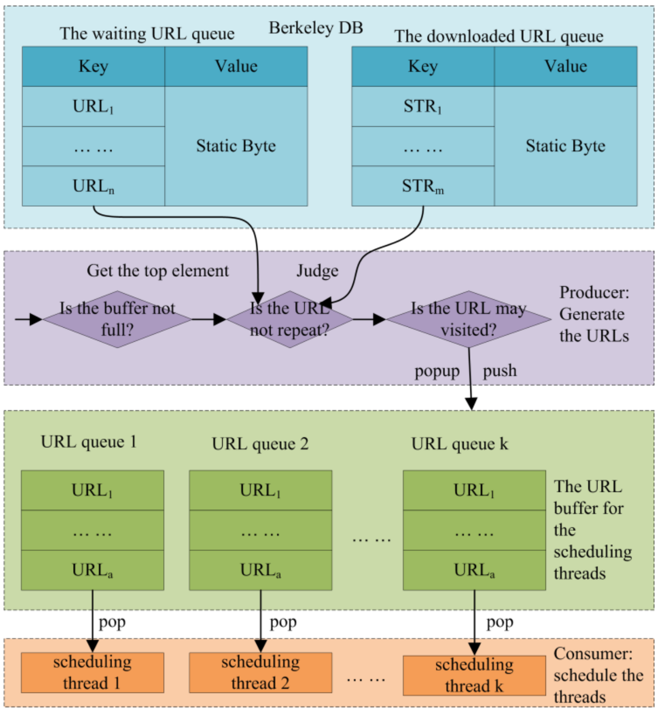

Fig.3. "Producer - Consumer" model

TABLE I. ALGORITHM 1

Algorithm 1: push($Queue, url$)

Function: Algorithm 1 pushes url into the URL queue for the scheduling thread.

Parameter Description: $Queue$, a collection of k URL queues for the scheduling threads. url, the URL to be pushed into the URL queue for the scheduling thread.

```
//Find out the shortest URL queue
$Queue_{shortest} = Queue_1$
For $i = 2$ to $k$ step 1
    If $Queue_i.length < Queue_{i-1}.length$ Then
        $Queue_{shortest} = Queue_i$
    End If
End For
//Push $url$ into the shortest URL queue
If $Queue_{shortest}.length < a$ Then
        pop( $QueueToDownload, url$ )
        push( $Queue_{shortest}, url$ )
End If
```

In Algorithm 1, it finds out the shortest URL queue for the scheduling thread through a loop "**For**" function, which is actually the first URL queue $Queue_{shortest}$ in the multiply shortest URL queues. For example, if there are three shortest URL queues, the lengths of these queues are all eight, and then $Queue_{shortest}$ is the first URL queue of these queues. After finding out $Queue_{shortest}$, if the length of $Queue_{shortest}$ is less than a, then the algorithm popups one URL from the waiting URL queue $QueueToDownload$ to url, and pushes url into $Queue_{shortest}$.

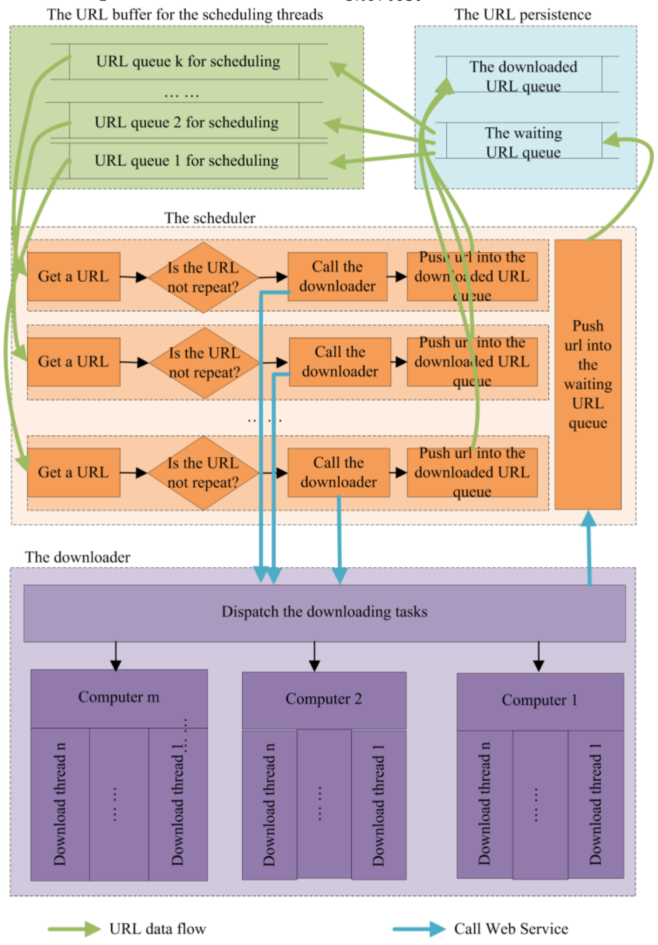

Fig.4. The designs of the scheduler and downloader

Algorithm 1 can judge the status of the URL queues. If the URL queue is not full, then the downloaded URL queue popups a URL into $Queue_{shortest}$. If we set $k = 20$, It shows the system has 20 scheduling threads. If we set $a = 10$, It shows the maximum length of the URL queue for scheduling thread is 10. If the length of each URL is 150 characters, then the size of the memory occupied by the URL buffer is the following value,

$$k \times a \times 150 \times 2 = 20 \times 10 \times 150 \times 2 = 60 \text{ K Byte} \quad (2)$$

Thus, the URL buffer only needs a little memory space, which avoids the memory overflow. Because of the URL buffer, it's efficiency close to memory storage situation when handling the URLs.

As showed in Fig.3, the "Consumer" in the "Producer-Consumer" model consists of k scheduling threads. The k scheduling threads respectively popup the URLs from the k URL queues for scheduling threads to consume the URLs.

C. Detachment of scheduler and downloader

According to the research described, we can see that the existing open-source software of the vertical crawler share the scheduler and the downloader in the same process or thread. It takes more time for downloading the web pages. Therefore, we consider a way to separate the scheduler and downloader. "Crawl while downloading" to "separate the downloader and the scheduler", the design idea is showed in Fig.4.

The "Producer" thread in the "Producer-Consumer" model continues to generate the URLs for the k URL queues for scheduling. The k URL queues for scheduling correspond to the k scheduling threads. In the scheduling thread, the first step is the repeated-URL judgment. If url is not downloaded, then the scheduling thread calls asynchronous the Web Service of the downloader. After calling asynchronously, the scheduling thread pushes url into the downloaded URL queue.

Another Web Service of the scheduler is used to receive calling by the downloader, then it parses out the URL array and pushes the URLs in the URL array into the waiting URL queue.

The design of the downloader is shown in Fig.5. The download threads of the downloader are distributed in multiple computers. When receive the request to download the URL, the downloader pushes the URL into the shortest URL queue for downloading. The download tasks will be dispatched to $m \times n$ download threads on m computers. Before the download thread finishes, the download thread tries to parse out the URL array by Jsoup, and returns the URL array to the scheduler by calling back the Web Service in the scheduler.

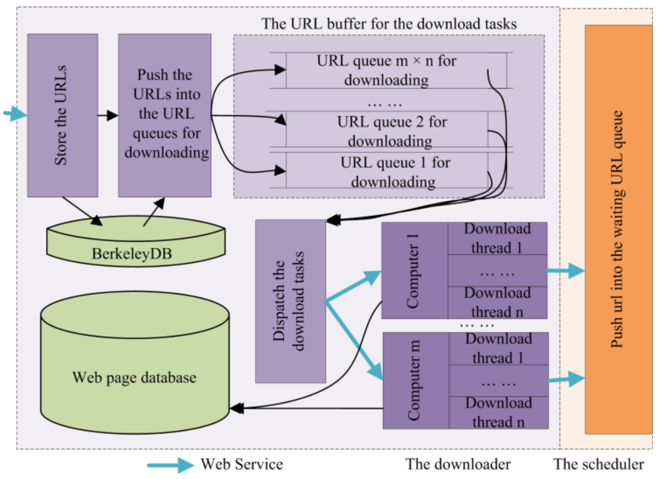
The URL buffer for the download tasks
Store the URLs
Push the URLs into the URL queues for downloading
URL queue m × n for downloading
… …
URL queue 2 for downloading
URL queue 1 for downloading
BerkeleyDB
Web page database
Dispatch the download tasks
Computer 1
Download thread 1
… …
Download thread n
… …
Computer m
Download thread 1
… …
Download thread n
Push url into the waiting URL queue
Web Service
The downloader
The scheduler

Fig.5. The architecture design of the downloader

Assuming that the average number of the characters in a URL string is l, and each queue stores a URLs, then the size of the download buffer may be considered as $l \times m \times n \times 2 \times a$. If 10 computers download the web pages, each computer runs 20 download threads, a URL has 150 characters in average, and each queue stores 10 URLs, then,

$$l \times m \times n \times 2 \times a = 150 \times 10 \times 20 \times 2 \times 10 = 600 \text{ K Byte} \quad (3)$$

Only 600 K byte space is needed for the URL buffer for the download tasks. If the capacity hard of the disk is sufficient, and then Berkeley DB can store the URLs with enough space with no worry about the problem of the memory overflow. The efficiency almost closes to directly use memory.

In the design of the downloader, there are two key problems. One is how to dispatch the URLs to the download thread. The algorithm is showed in Table II. The other is how to dispatch the download tasks. The algorithm is showed in Table III.

TABLE II. ALGORITHM 2

Algorithm 2: pushURLToQueue($Queue, url$)

Function: Algorithm 2 pushes url into the URL queue for downloading

Parameter Description: $Queue$, the URL buffer for download tasks, a set of $m \times n$ queue; url, a URL to download

```
//Find out the shortest URL queue
$Queue_{shortest} = Queue_1$
For $i = 2$ to $m \times n$ step 1
    If $Queue_i.length < Queue_{i-1}.length$ Then
        $Queue_{shortest} = Queue_i$
    End If
End For
//Push $url$ into the shortest URL queue
If $Queue_{shortest}.length < a$ Then
    pop( $QueueToDown, url$ )
    push( $Queue_{shortest}, url$ )
End If
```

TABLE III. ALGORITHM 3

Algorithm 3: dispatchDownLoadTask($Queue$)

Function: Algorithm 3 popups a URL from the URL buffer for the download tasks, and dispatches the URL to the download thread.

Parameter Description: $Queue$, the URL buffer for download tasks, a set of $m \times n$ queue.

```
//Find out the longest queue
$Queue_{longest} = Queue_1$
**For** $i = 2$ **to** $m \times n$ **step** 1
    **If** $Queue_i.length > Queue_{i-1}.length$ **Then**
        $Queue_{longest} = Queue_i$
    **End If**
**End For**
//dispatch a URL from the longest URL queue to the download thread
pop( $Queue_{longest}, url$ )
$computerNo = \mathbf{Int}(i/n)$
$threadNo = i\%n$
executeDownTask($computerNo, threadNo, url$)
```

Algorithm 2 finds out the shortest URL queue $Queue_{shortest}$ for download task through a loop "**For**" function. If the length of $Queue_{shortest}$ is less than a, then the algorithm popups a URL from the queue $QueueToDownLoad$ in Berkeley DB to url, and pushes url into $Queue_{shortest}$.

Algorithm 3 finds out the longest URL queue $Queue_{longest}$ for download task through a loop "**For**" function. Next, the algorithm popups a URL from $Queue_{longest}$ to url. $\mathbf{Int}(i/n)$ is used to calculate the number of the computers $computerNo$, and $i\%n$ is used to calculate the thread number $threadNo$ in the computer $computerNo$ participating in the download tasks. Finally executeDownTask() method calls the Web Service on computer $computerNo$ to execute the thread $threadNo$ and download the content of url.

D.Implementation of scheduler and downloader

The design ideas of the scheduler and the downloader are inspired by Aspect-Oriented Programming (AOP) and Dependency Injection (DI) of Spring [23, 24]. The design of the scheduler is shown in Fig.6, and the downloader is designed in the similarly way.

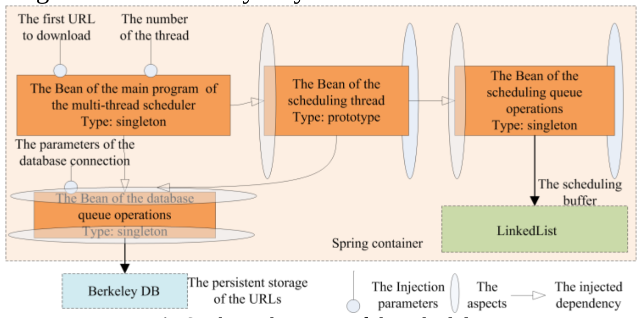
The first URL to download
The number of the thread
The Bean of the main program of the multi-thread scheduler Type: singleton
The Bean of the scheduling thread Type: prototype
The Bean of the scheduling queue operations Type: singleton
The parameters of the database connection
The Bean of the database queue operations Type: singleton
The scheduling buffer
LinkedList
Spring container
Berkeley DB
The persistent storage of the URLs
The Injection parameters
The aspects
The injected dependency

Fig.6. The architecture of the scheduler

Through using the DI features of Spring, we could apply the context configuration file of Spring, and inject the configuration parameters and the dependencies of the Bean into the container of Spring. The Bean of the main program of the multi-thread scheduler is injected the first URL "$firstURl$" to download and the number "k" of the thread. The Bean of the database queue operations is injected the parameters of the database connection including the storage address of the data file, the buffer size, the flag of transaction, etc.

The Bean type of the database queue operations is singleton. It shows the container of Spring only maintains one Bean of the database queue operations. The database connection can be established in the Bean initialization, need not to be opened and closed every time. The same Bean of the database queue operations is depended on multiple Beans [24].

Considering the program code of each scheduling thread is same, the Bean type of the scheduling thread is set as prototype. After the main program of the multi-thread scheduler starts, it will generate one by one the Beans of the scheduling threads. The life cycles of these Beans are managed by the container of Spring. All of the scheduling threads share the Bean of the scheduling queue operations and the Bean of the database queue operations.

Through using the AOP features of Spring, along with the methods of the Beans, we can inject pre-Advice and post-Advice for logging, parameter verification, etc. Though recording the logs, we know the store the time consumption and operation event, compare the performance with other crawls, monitor and improve the performance of MCMTCrawler.

IV. EXPERIMENTAL EVALUATION

In this section, we present an experimental evaluation of our MCMTCrawler. To examine the performance of MCMTCrawler, we compare it with three representative crawlers, as listed in Table IV. The first crawler is a single-computer and single-process crawler. The second crawler is a single-computer and multi-thread crawler. The third crawler is also an open source multi-thread crawler called Crawler4j.

Fig.7 shows the comparison of crawling speed of four different crawlers. The basic idea is to observe the average execution time of downloading 100 URLs under different crawling tasks, the number of web pages in different crawling tasks increases from 10,000 to 600,000. From the figure, it is very easy to find that our MCMTCrawler has the best speed than other three crawlers. More especially, we can get the following remarks from Fig.7.

(1), the performances of four crawlers are relatively stable in different downloading tasks. As the number of downloading web pages increases, the average execution time required is slightly increased.

(2), the speed of normal single-computer and multi-thread crawler is very close to that of Crawler4j, thus they have almost equal efficiency.

(3), as showed in Fig.7, the speed of single-computer and multi-thread crawler (Crawler4j) is 2 times as the single-computer and single-process crawler, while the speed of our MCMTCrawler is 5 times as the single-computer and single-process crawler. Furthermore, in the crawling task of 600,000 web pages, the single-computer and single-process crawler takes about 1.43 days, the normal single-computer and multi-thread crawler takes 0.82 days, Crawler4j needs 0.83 days, and our MCMTCrawler only takes 0.28 days(about 6.83 hours), as shown in Fig.8.

TABLE IV. THE DESCRIPTION OF FOUR DIFFERENT CRAWLERS

Name	Description of Crawler	Environment of Hardware	Environment of Network
Single-computer and single-pro_cess crawler	Scheduler and downloader in a same process	ThinkPad T460s，Intel(R) Core(TM) i5-6200U CPU @ 2.30GHZ，8G RAM，512G SSD，Windows 10	The speed of LAN is 1000 Mbps, the speed of WAN is 200 Mbps
Single-computer and multi-thread crawler	Scheduler and downloader in a same process, one process contains 20 threads	ThinkPad T460s，Intel(R) Core(TM) i5-6200U CPU @ 2.30GHZ，8G RAM，512G SSD，Windows 10	The speed of LAN is 1000 Mbps, the speed of WAN is 200 Mbps
Crawler4j	Scheduler and downloader in a same process, one process contains 20 threads	ThinkPad T460s，Intel(R) Core(TM) i5-6200U CPU @ 2.30GHZ，8G RAM，512G SSD，Windows 10	The speed of LAN is 1000 Mbps, the speed of WAN is 200 Mbps
MCMTCr_awler	Scheduler and downloader are separated, the number of the computer participating in the download tasks is 3, the number of the thread of each computer participating in the download tasks is 20	**Scheduler**：ThinkPad T460s，Intel(R) Core(TM) i5-6200U CPU @ 2.30GHZ，8G RAM，512G SSD，Windows 10 **Downloader**：Dell R730 E5-2603V4，64G（16G*4）RAM，256G*2 SSD+4T 3.5 SAS 7.2K*3，RAID5，Windows 10	The speed of LAN is 1000 Mbps, the speed of WAN is 200 Mbps

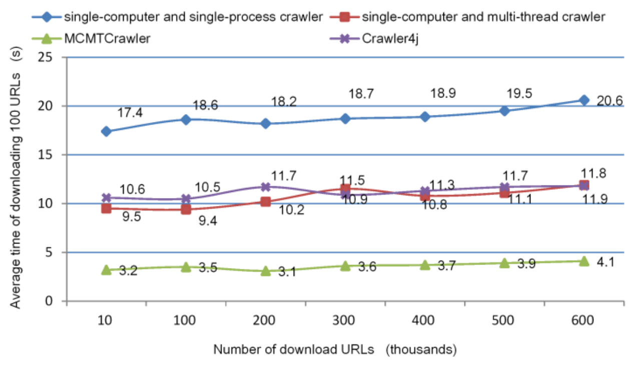

Fig.7. The comparison of crawling speed

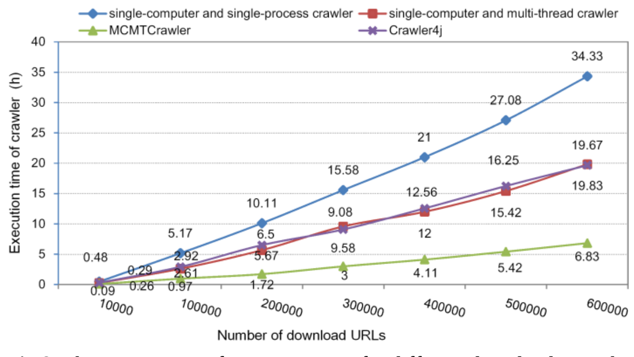

Fig.8. The comparison of execution time for different downloading tasks

After experimentation, we also find out that if the number of the threads for accessing to Berkeley DB is too much, then Berkeley DB will generate some write-locks. It may cause the system to be unable to work. Berkeley DB is a relatively simple Key-Value database, has limited capacity for concurrent access. After our testing, the number of the threads for accessing to Berkeley DB can be set to a value less than or equal 30.

According to the extensive experimental results, we find that it helps little improving the speed of a crawler by increasing the thread number in a single computer. The main reason is that the network bandwidth of a single computer is limited. Therefore, it is a valid method to significantly improve the crawler speed by using multiple computers to execute the crawling tasks at the same time. However, due to the limitation of Internet bandwidth, there is a bottleneck in the speed of the crawler. Otherwise, due to the limitation of the experimental environment and software program development, we don’t use the distributed development in this paper. To further enhance the speed of our MCMTCrawler, we can take the following ways.

(1), the first way is to increase the number of computers to download the web pages, thus increasing the concurrency degree.

(2), the second way is to use the ideas of distributed design for mapping different downloading tasks into different Internet environment.

V.CONCLUSIONS

To quickly crawl the web pages of a special large commercial website, a new crawler is designed, called MCMTCrawler. MCMTCrawler uses Berkeley DB and simple Key-Value structure for persistent storage. The MD5 algorithm is employed to map a URL to a 32-length string for improving the speed of reading-writing and avoiding the problem of memory overflow. MCMTCrawler takes advantage of the "Producer-Consumer" model to speed up the distribution and processing of URLs. With the help of Spring-based AOP and DI ideas, MCMTCrawler designs the scheduler and the downloader separately. Three algorithms are developed, which implement the functions including "how to put a URL into the waiting URL queue of scheduler", "how to put a URL into the waiting URL queue of downloader", and "how to start a download task from the waiting URL queue". The experimental results show that, under the three computers for downloading, the speed of MCMTCrawler is 5 times than the single-computer and single-process crawler, and is 3 times compare to the single-computer and multi-thread crawler Crawler4j. Moreover, MCMTCrawler only takes 0.28 day to download 600,000 web pages.

In the subsequent work, we will further improve the crawling speed of MCMTCrawler from the distributed architecture. At the same time, we also will carry out the research of the semantic analysis of large-scale commercial website, including the design of semantics Web Ontology Language (OWL), the special analysis and inference software, the comparison and analysis of the commodity and price, the content and data analysis of web pages, etc.

References

[1] R.Chandy, M.Hassan, and P.Mukherji, Big Data for Good: Insights from Emerging Markets. *Journal of Product Innovation Management*, vol.34, no.5, 2017, pp.703-718.

[2] Pan.W, Yang.Q, Aggarwal.C, and Koch.C, Big Data. *IEEE Intelligent Systems*, vol.32, no.2, 2017, pp.7-8.

[3] Kimura. F, Urae. H, Tezuka. T, Maeda. A, Multilingual Translation Support for Web Pages Using Structural and Semantic Analysis, *IAENG International Journal of Computer Science*, vol.39, no.2, 2017, pp.276-285.

[4] Agre.G.H and Mahajan.N.V, Keyword focused web crawler, *In Proceedings of the 2nd International Conference on Electronics and Communication Systems (ICECS)*, Coimbatore, India, Feb 26-27, 2015, pp. 1089-1092.

[5] Samita.B, Sharaf.H and Khoja.S, A Framework for Focused Linked Data Crawler using Context Graphs, *In Proceedings of the 2015 International Conference on Information & Communication Technologies*, Karachi, Pakistan, Dec 12-13, 2015, pp.1-6.

[6] Anish.G and Priya.A, Focused Web Crawlers and Its Approaches, *In Proceedings of the 1st International Conference on Futuristic trend in Computational Analysis and Knowledge Management*, Noida, India, Feb 25-27, 2015, pp.619-622.

[7] Nisha.P, Rajeswari.K, Joshi.A, Implementation of an Efficient Web Crawler to Search Medicinal Plants and Relevant Diseases, *In Proceedings of the 2016 International Conference on Computing Communication Control and automation*, Pune, India, Aug 12-13, 2016, pp.1-4.

[8] Siddesh.G.M, Suresh.K, Madhuri.K.Y, Nijagal.M, Rakshitha.B.R and Srinivasa.K.G, Optimizing Crawler4j using MapReduce Programming Model, *Journal of The Institution of Engineers*, vol.98, no.3, 2016, pp.329-336.

[9] Muqrishi.A.A, Sayed.A, and Mohammed.K, Caseng: Arabic semantic search engine, *Revista Chilena De Historia Natural*, vol.73, no.2, 2015, pp.643-651.

[10] Xia.J, Wan.W, Liu.R , Chen.G, and Feng.Q, Distributed web crawling: A framework for crawling of micro-blog data, *In Proceedings of the 2015 International Conference on Smart and Sustainable City and Big Data (ICSSC)*, Shanghai, China, July 26-27, 2015, pp.62-68.

[11] Gao.Q, Xiao.B, Lin.Z, Chen.X.Y, and Zhou.B, A high-Precision Forum Crawler based on Vertical Crawling, *In Proceedings of the 2009 International Conference on Network Infrastructure and Digital Content*, Beijing, China, Nov 6-8, 2009, pp.362-367.

[12] Zhou.B, Xiao.B, Lin.Z, and Zhang.C, A Distributed Vertical Crawler Using Crawling-Period Based Strategy, *In Proceedings of the 2010 International Conference on Future Computer & Communication, Wuha*, China, May 21-24, 2010, pp.306-311.

[13] Dajie.G and Zhijun.D, A Task Scheduling Strategy based on Weighted Round-Robin for Distributed Crawler, *In Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, London, UK, Dec 8-11, 2014, pp.848-852.

[14] Yogesh.S.K and Nagesh.D.K, An Efficient Deep Web Harvesting Based on Two Stage Crawler, *International Journal of Engineering Science and Computing*, vol.6, 2016, pp.2473-2476.

[15] Feng.Z, Jingyu.Z, Chang.N, Heqing.H, and Hai.J, SmartCrawler: A Two-Stage Crawler for Efficiently Harvesting Deep-Web Interfaces, *IEEE Transcations on Services Computing*, vol.9, no.4, 2016, pp.608-620.

[16] Hatzi.V, Cambazoglu.B.B, and Koutsopoulos.I, Optimal Web Page Download Scheduling Policies for Green Web Crawling, *IEEE Journal on Selected Areas in Communications*, vol.34, no.5, 2016, pp.1378-1388.

[17] Seyed.M.M, Gregor.V.B, Guy-Vincent.B, and Iosif.V.O. GDist-RIA Crawler: A Greedy Distributed Crawler for Rich Internet Applications, *Lecture Notes in Computer Science book series*, vol.8593, 2014, pp.200-214.

[18] Ayar.Pr and Sandip.C, Efficient Focused Web Crawling Approach for Search Engine, *International Journal of Computer Science and Mobile Computing*, vol.4, no.5, 2015, pp.545-551.

[19] Alqaraleh.S, Ramadan.O, and Muhammed.S. Efficient watcher based web crawler design, *Aslib Journal of Information Management*, vol.67, no.6, 2015, pp.663-686.

[20] Emin.I.T and Bedirhan.U, WIVET-Benchmarking Coverage Qualities of Web Crawlers, *Computer Journal*, vol.60, no.4, 2017, pp.1-21.

[21] Seyfi.A, Patel.A and Joaquim.J.C, Empirical evaluation of the link and content-based focused Treasure-Crawler, *Computer Standards & Interfaces*, vol.44, 2016, pp.54–62.

[22] Boonkrong. Sirapat, Somboonpattanakit. Chaowalit, Dynamic salt generation and placement for secure password storing, *IAENG International Journal of Computer Science*, vol.43, no.1,2016, pp.27-36

[23] Ziyun.D, Jing.Z, and Tingqin.H. Automatic Combination Technology of Fuzzy CPN for OWL-S Web Services in Supercomputing Cloud Platform. *International Journal of Pattern Recognition and Artificial Intelligence*, vol.31, no.07, 2017, pp.1-27.

[24] Iuliana.C, Rob.H, Chris.S, and Clarence.H, Introducing Spring AOP, *Pro Spring 5*, Apress, Berkeley, CA, 2017.