Part A: Collect your data and Index with Lucene

Project target:

Crawl the Wiki Web page using jsoup.

Crawling system

**1. Architecture**

The crawling system mainly including three parts: the crawling part, writing part, the database part. These three parts works in pipeline: crawling part crawls the wiki web page; the writing part write the result of crawling part into the database; the database part saves all the web page data that have been crawled. To make three parts work in pipeline, we need two interconnection parts: one LinkedBlockingQueue is used between crawling part and writing part; the database SQL connection is used between writing part and database part. Overall the basic architecture of our crawling system is shown in Figure 1.



Figure 1 Basic architecture of crawling system

Wiki crawlers spend a lot of time waiting for responses to requests. To improve the efficiency of our crawling system, both crawling part and writing part use threads: CrawlThread and WriterThread. So each CrawlThread, WriteThread and corresponding LinkedBlockingQueue and SQL connection form one pipeline to pour data into same database. So the full architecture of our crawling system including many pipeline shown in Figure 1, and all pipeline connect to the same Database, and it is shown in Figure 2.



Figure 2 Full architecture of crawling system

**2. The Crawling Strategy**

**2.1 Strategy: How crawler thread work**

In our crawler, the general search algorithm is used. And the main content of the crawler thread can be written in following pseudo-code. In our algorithm of crawler, we limited page depth to 10. Because higher page depth will reduce the efficiency of crawler.

**Start** its writer thread;

Get *seed* from entry;

Add Seed to *Queue*;

**while** Count< setting numbers **do**

**if** (*Queue* is empty) **do**

Get seed from entry;

Add Seed to *Queue*;

Get URL from *Queue*;

Request the URL and get web page;

Add URL to *visited set*;

Process web page;

Save processed web page into the linked blocking queue to writer;

**if** (Depth of this web page is larger) **do**

**continue;**

**else do**

Get all URLs from processed web page

**for** URL in URLs **do**

**if** URL not in *visited set* and URL is from Wiki **do**

Add URL to *Queue*;

**end**

**2.2 Strategy for seed and duplicated pages**

To make the crawler work, we need a set of seeds or a set of URLs of Wiki web page. So each thread will first request this entry URL [*https://en.wikipedia.org/wiki/Special:Random*](https://en.wikipedia.org/wiki/Special:Random) for the seed. This is the random page provided by Wiki server, and it will return some Wiki page randomly. And the returned Wiki page is served as the seed for the crawler. In addition, to make the crawler more politeness, the CrawlThread will sleep for 5 seconds after every crawling.

To avoid duplicated URL being crawled again, there is one <set> called vistedURL is shared by all CrawlThread. This <set> saves the URL of all web page that have been crawled and saved. To improve efficiency and maximize the throughput, here we use database to save all the webpage data. The database used here is SQLITE.

**2.3 Strategy: how to process the web page**

One of the important parts here is how to process the web page, and all the page process is coded in method process () in CrawlThread class. For almost all Wiki web page, there has one unique title; the main content, which is made of many paragraphs; and some words represent the category of this page. So, for every web page, we first split it into these three parts. And this is done by following three methods provided by jsoup.

*Element elTitle = doc.getElementById("firstHeading");*

*Element elContent = doc.selectFirst("#mw-content-text .mw-parser-output");*

*Element elCategory = doc.getElementById("mw-normal-catlink");*

a) Title

As the title is unique for all Wiki web pages, the title is used for the primary key when saving to SQLITE database. Using title as primary key has two advantages than using URL: one is that primary key is shorter than URL; the other is that URL can be recovered from title when necessary.

The recovery process can be this:

URL = *"*<https://en.wikipedia.org/wiki/>*"* + title

For instance, the URL of web page with the title Computer is:

<https://en.wikipedia.org/wiki/Computer>

For title with more than one work, we need change the space to *"\_"* in title. And then follows the recovery process. For instance, the URL of of web page with the title Computer science is:

<https://en.wikipedia.org/wiki/Computer_science>

b) Content

For the main content, we can simply convert the element to string using text() and trim() method.

*String title = elTitle.text().trim();*

But it will generate references, edit links, spaces, \r\n, tables, and captions that is not belong to the main content. In order to make the index part more efficiency, we need remove these parts from the main content. To remove these, we use several methods provided by jsoup.

*elContent.select("sup[class='reference']").remove();*

*elContent.select("span[class='mw-editsection']").remove();*

*Arrays.asList("table", "div").forEach(tag -> elContent.select(tag).remove());*

c) Category

For the words in category, we covert it to List<String>. And then saved into database.

**2.4 Strategy: how to write the web page class into database.**

In our crawling system, one crawl thread will have one write thread to write the web page generated into the SQLITE database. In order to improve the efficiency and throughput, the write thread use batch. Every 50 records form one batch, and they are written at one time. In order to concurrent the crawl thread and write thread, the page queue from crawl thread to write thread is blocked queue.

Here is a quick view of the result of our database for crawler:

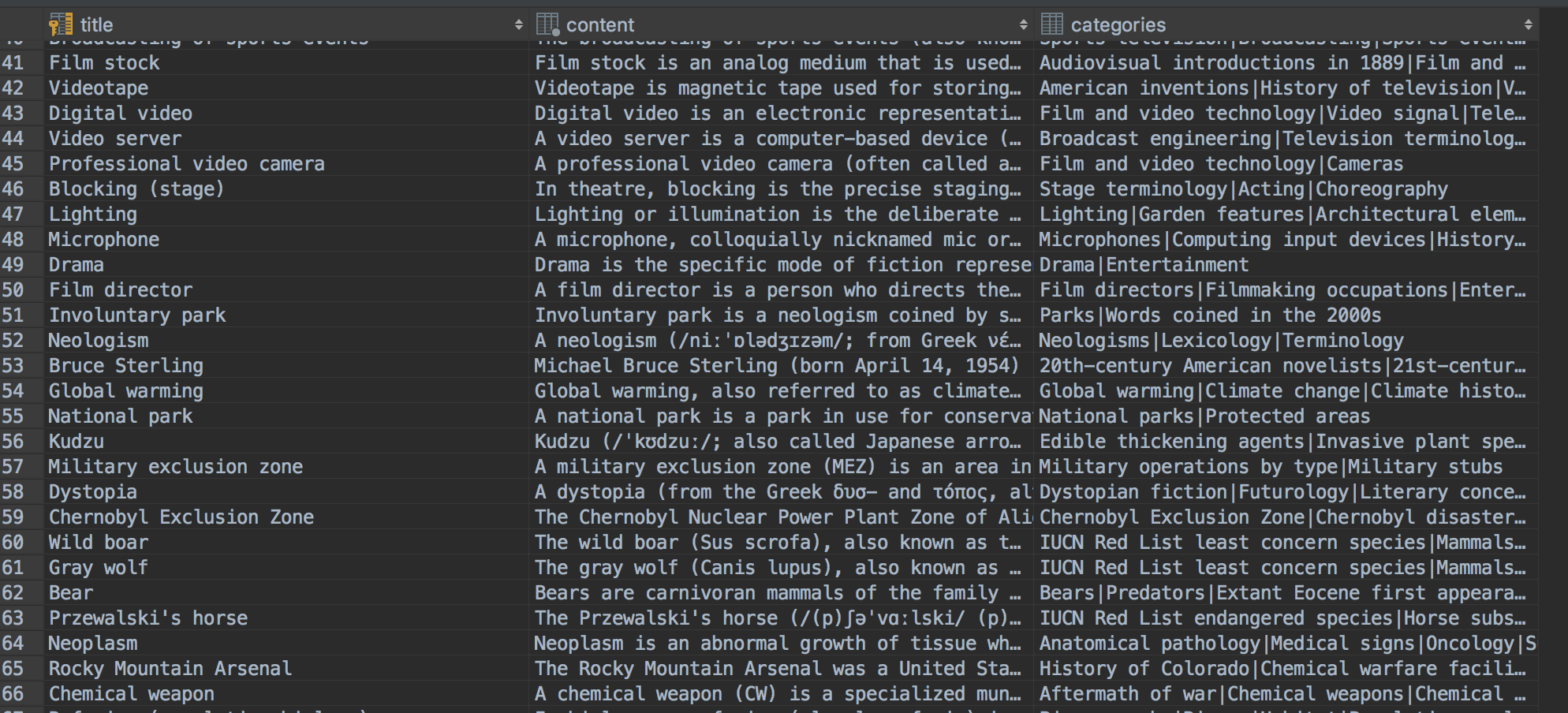


Figure 3 Result in SQLITE database

**2.5 Some Obstacles and solutions during coding**

One big obstacle we encounter during coding is how to only the main content of the web page. In Wiki page, there are many other parts other than the main content, such as the picture, table, edit hyperlink, reference link etc. If we do not remove these part from our main content text. There will be many irrelevant words and chars. So we spent a lot of time on this part. And it is handled using the method mentioned in 2.3(b).

**3. Limitation of the crawling system**

There is one limitation in our crawling system: the consistency and corresponding efficiency. Although the visited URL <set> is used to avoid to search duplicated web page, the sequential consistency is still not ensured, because the visited URL<set> is not saved persistent with the database. Think about such case: Firstly, crawl the Wiki for some time, and stop; After some time, re-run the Crawler; when re-run the Crawler, the visited URL <set> is empty, and it is not consistent with the database.

How to solve this, we ensure that this crawling system is eventually consistency by checking the page queue before write it database. This is done by the SQL statement: "INSERT **OR IGNORE** INTO pages". Therefore, every Wiki page in the database will be unique. But this solution is not very good. Because when we re-run the Crawler, it may crawler some pages that already saved in database before this run, and this reduce the efficiency of our crawling system.

**4 Instruction on how to deploy the crawler**

In our crawling system, we create one class called WikiCrawler as the start point of our system. It contains one constructor as follows:

*public* ***WikiCrawler****(int numOfThreads, int numOfPages, int crawlDepth, int crawlInterval, String entryUrl, String crawlHostRegex, String crawlPathRegex,String jdbcUrl)*

To use command line to input argument, "commons-cli-1.4.jar" is used in our system. When using *WikiCrawler* in command line, we need give following argument:

Table 1 Argument table

|  |  |  |
| --- | --- | --- |
| **Short Name** | **Argument Name** | **Default Value** |
| *t* | *numOfThreads* | 10 |
| *c* | *numOfPages* | (this value will ensure 5GB data)750000 |
| *d* | *crawlDepth* | 10 |
| *i* | *crawlInterval* | 500 |
| *u* | *entryUrl* | "https://en.wikipedia.org/wiki/Special:Random" |
| *H* | *crawlHostRegex* | "^en.wikipedia.org$" |
| *P* | *crawlPathRegex* | "^/wiki/[^:]\*$" |
|  | ***jdbcUrl*** | **Required parameter no default value** |
| *l* | *FILE NAME* | STDOUT |

**Lucene indexing strategy**

When building Lucene index, we also run multi-threaded batch jobs to improve the performance of our system. When acquiring raw content, we use batch read to fetch records from our database. Intuitively, we chose the batch factor as 50 to make a balance between reducing I/O and not consuming too much main memory.

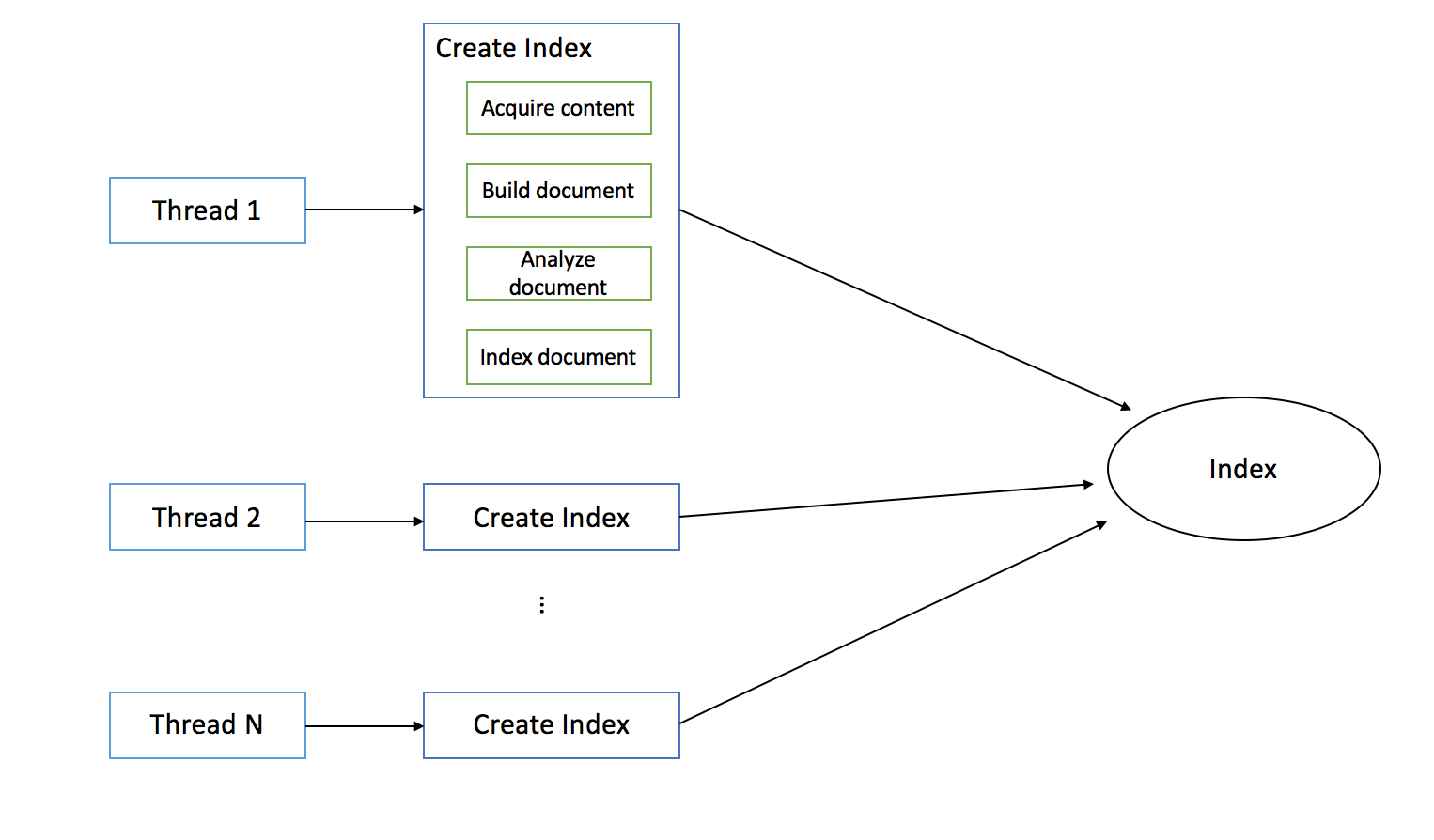


Figure 4 Architecture of Lucene Indexer

We make one webpage as a document. To make it easier to maintain, we make each document has the three fields: title, content and categories, which are exactly the three attributes stored in our database. Since we want to search title, content and categories later, we build indexes for all of them, but in order to save some space, only title will be stored in the index and if user need to view the content or categories, the system will use the title to fetch them from our database.

As for the analyzer part, since we already remove most of the irrelevant part and get the plain text while crawling the webpage as discussed in section 2.3(b), the analyzer part is pretty simply and straightforward. We use a separate analyzer to analyze categories since when stored in the database we use the character “|” to separate different categories so that the categories field has one more special stop world “|”. After that, we just use standardAnalyzer to analyze the documents.

We report the process when indexing every 1000 pages. As is shown in Figure 4, we finished indexing totally 744723 pages with in 3 hours and 20 minutes.



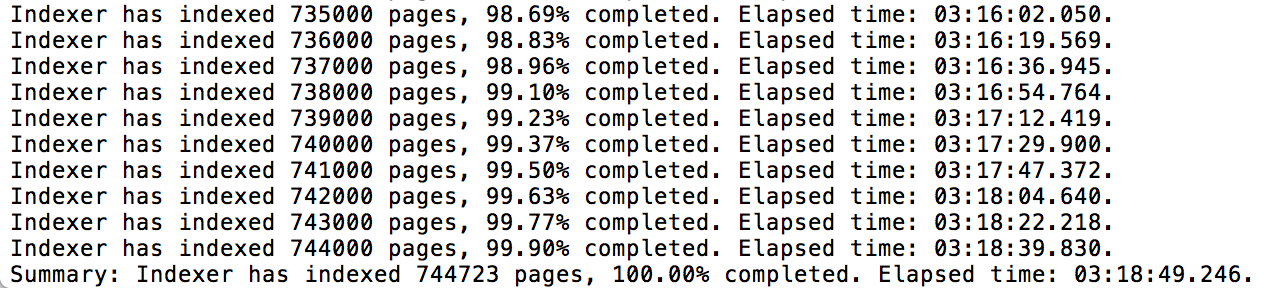


Figure 4 snapshot of process report