Indexer and Query Processor

Introduction

This project consists of two parts: An Index Generator that creates inverted index structures from a set of downloaded pages; and a Query Processor that uses the index created in the previous part to answer queries typed in by a user. Both parts are written in Java with some Shell scripting in the first part. Detailed implementation analysis will be discussed in the following sections.

Modules

1. Index Generator

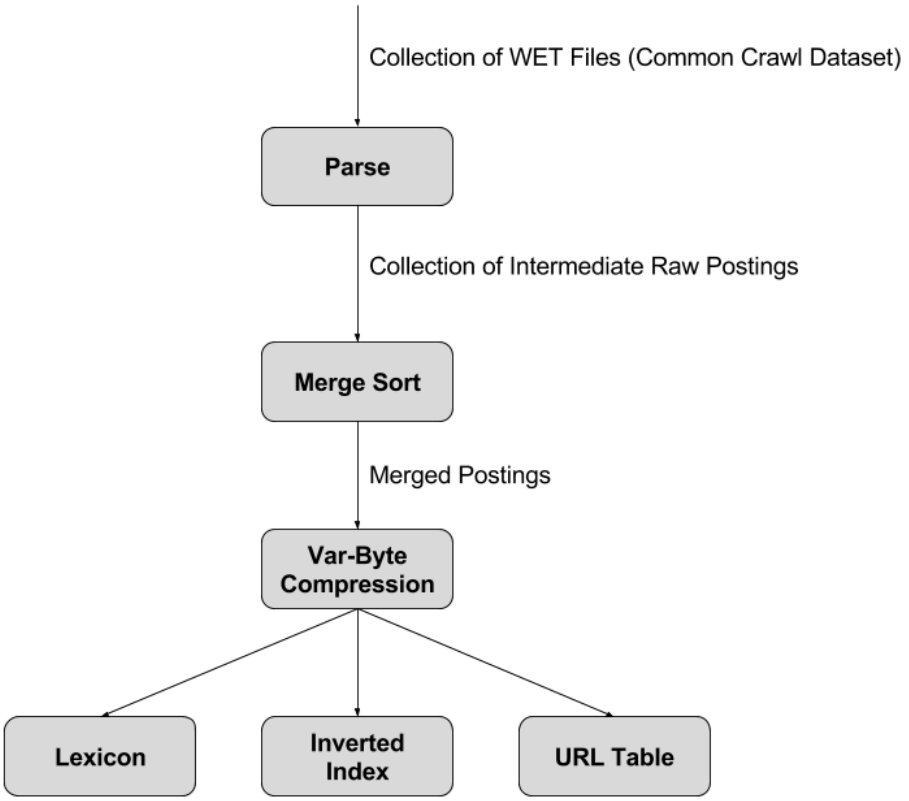

Figure 1 Index Generator Flow

Index Generator creates 3 files as part of Inverted Index Structure, described below:

- ***URL Table***: Has all the URL's crawled. Document ID's are ordered by time of crawl *Format*: {*Document-ID*, *URL*, *Length of Document*}

- ***Inverted Index***: Contains postings for words arranged in Chunks and Blocks (See *Appendix 2* for detailed explanation of Chunks and Blocks)
  Chunk Format: {Total Blocks, [Last Doc Ids], [Size of Blocks], Block-1…Block-N}

    - Total Blocks – Number of blocks(uncompressed) in a Chunk
    - [Last Doc Ids] – Last Document Ids (uncompressed) of all blocks in chunk
    - [Size of Blocks] – Size (in bytes and uncompressed) of all blocks in chunk

  Block Format: {Size of Doc-Ids, Size of Freqs, [128 Doc Ids], [128 Freqs]}
    - Size of Doc Ids – Size of all 128 Document Ids in a block (uncompressed)
    - Size of Freqs – Size of all 128 Frequencies in a block (uncompressed)
    - Doc Ids and Freqs are in compressed form (See Appendix 3 for details)

- ***Lexicon***: Has all the words with metadata to find the corresponding inverted-list
  Format: {Word, Chunk Num., Block Num., Posting Num., Total Postings, Span}
    - Word – String word extracted from web-pages.
    - Chunk Num. – Start Chunk number where this word's inverted list starts
    - Block Num. – Block number in the chunk where inverted list starts
    - Posting Num. – Posting number in the chunk where inverted list starts
    - Total Postings – Total Word Postings in the inverted list for this word
    - Span – Total Chunks the inverted list for this word spans

Main Components of Index Generator

- *Intermediate Postings Generator (Java class)*
  Main class that reads in WET files ([1] describes the file format in more detail), parses them and creates Raw Intermediate Postings.
- *Parser (Java class)*
  Parses the HTML Files and WET files format and filters words from them.
- *Buffered File (Java class)*
  Buffer backed file, which dumps the contents of a buffer to a new file when full. This file abstracts from the main class, the details of creating intermediate postings file. When a buffer is full, a new posting file is created and all the words in that file are sorted. Format of this intermediate file is below (see Appendix 1 for details on Word Posting)

  {Word, WordPosting-1, WordPosting-2 … Word-Posting-n}

  There are 2 *Buffered Files*. One for dumping the contents of Word Postings and one for URL Table. Buffer size for each can be set according to a properties file as described later sections.

- *Postings Merger (Shell Script)*
  Postings Merger is responsible for merging all the sorted files generated by Intermediate Postings Generator.
- *Index Creator (Java class)*
  Main class that reads in the merged postings file in the above step and generate the inverted-index structures in the format discussed above.
- *Buffered Inverted List (Java class)*
  Follows the same concept as of *Buffered File class* (described above) with the difference that when a buffer is full, it does not create a new file but appends to the same file. Also takes care of Chunks and Blocks format.
- *Chunk (Java class)*
  Manages the lifecycle for a chunk i.e. keeps track of the current chunk size and raises a flag when that chunk is full. Chunk size is configurable
- *Block (Java class)*
  Manages the lifecycle for a block i.e. keeps track of when a block is full and raises flag appropriately. Each block limit is exact 128 postings. Keeps the postings in compressed form. So, Block is responsible for postings compression (var-byte compression)

Detailed Workflow for Index Generator

Figure-1, gives a bigger picture to Index Generator flow. We explain each component in detail in this section.

- *Parsing of HTML and WET Files format*
  For HTML we use the *Poly-Parser (Java version)* [2]. For WET Files, we use the parser provided by the *Common Crawl Community* [3]. Even though this parser is designed for reading WARC format, we made one change in the source code to adapt it to read WET files too. Both these parsers will give us an *array of words.* We use this to associate words with doc. ids.
- *Document ID assignment*
  We don't sort a document by its URL, but instead we assign document ids to documents in the order they are read and parsed by the above parsers.
- *Intermediate Postings Generation*
  After parsing a document, we send all its postings to a *Buffered File*. URL of the same document is sent to another *Buffered File*. As soon as the buffer is full, a new intermediate file (by name – *intermediateposting_0...0*) is generated. This is the most compute intensive part of Index Generator.
- *Intermediate Postings Merging*

We use the UNIX *join* functionality to merge all the sorted files. These files are merged on the actual words. This is done using a shell script that automates this process of merging.

- *URL Table Merger*
  Since all the URL's are already in order by Document Id, we simply append all the intermediate URL table files using UNIX *cat* functionality.
- *Index Generation*
  After merging all postings to a single file, we arrange the words and its inverted list into chunks and blocks. Buffer size is configurable and is by default set to be equivalent to size of 100 chunks. So we first create 100 chunks in memory and then dump them to inverted-index file. (See Appendix 3 for details on the compression used in this part)
  Remember that now words and its metadata go into the lexicon file while the actual word-postings to the inverted index file. URL Table was already created in the above step, so no more work there. After this step we have out inverted-index structures (Lexicon, URL Table and Inverted Index)

Performance and Size of Intermediate and Final Structures

- Total WET Files = 100
  15.1 GB GZipped, downloaded by following instructions from [4]
- Total Intermediate Files generated = 1114
  18.1 GB on disk, 6.1 hours for files generation
- Total Intermediate URL Table Files = 17 (484 MB on disk)
- Total time for merging all intermediate files = 3.21 hours
- Total Size of merged file = 17.2 GB on disk
- Total Size of final Inverted Index (compressed) = 4.17 GB
- Total Size of Lexicon = 990 MB
- Total Size of URL Table = 484 MB
- Total time for final index generation = 34.5 minutes
- Total Documents in Index = **5.3 million**
- Total Words in Lexicon = **30.1 million**

Limitations

- Unix Merge does not work on binary data. So having binary data or compression in intermediate steps was not possible.
- While merging intermediate postings (merge-sort functionality) we need to run Postings Merger program multiple times. Suppose we have 1000 files to merge, then it is more efficient to perform the merge in multiple steps rather than merging all at once. Current program does not support such automatic mini-merges.

2. Query Processor

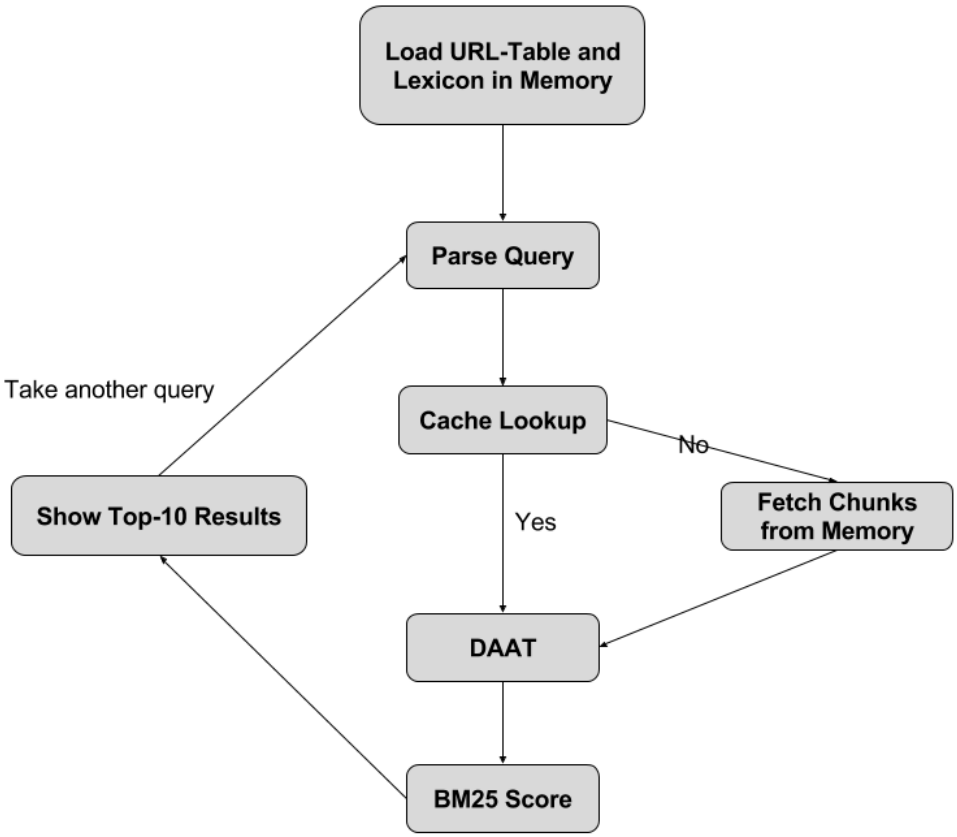

Figure 2 Query Processor

Main Components of Query Processor

- *Query Processor (Java class)*
  Responsible for loading the URL Table and Lexicon Structures into memory along with asking user for query and processing it.
- *Inverted List for Word (Java class)*
  Represents an inverted list for a word. Manages all the blocks for a which an inverted list spans. This class is where the cache and disk access happen This class also has static methods for *openList(word), closeList(list), nextGEQ(list, did), getFreq(list).* See Appendix 4 for details on this interface
- *Memory Chunk (Java class)*
  Manages a Chunk concept. As soon as a chunk is read, this class extracts its metadata and manages all the blocks inside it.

- *Memory Block (Java class)*
  Manages a Block concept. Once a block is read, this class extracts all its metadata and aligns pointer to position where next byte should be read.
- *Word Metadata (Java class)*
  POJO class for word metadata as stored in the lexicon.
- *Cache (Java class)*
  FIFO Implementation for the Cache. Chunks are cached. We use a dynamic cache over static cache. Cache size is configurable. By default, it is 25 MB.

Detailed Workflow for Query Processor

- *Looking up in Lexicon for all Words*
  When a query is entered, Query Processor searches for a match in Lexicon, if for even one word no match is found that query is ignored.
- *Evaluating Conjunctive and Disjunctive Queries*
  Document at a time query processing technique is used to answer user-queries. Appendix 4 lists the primitives used to implement such strategy. Appendix 4 gives a high level view of such primitives. We explain in detail how it was implemented in the project.
    1. *openList*
        a. Takes a WordMetadata Object
        b. Gets all the chunks for the word and checks if they are present in the Cache. If not, then extracts the right chunks. We use the *Java Random Access File* feature to fetch a particular chunk in any part of the file.
        c. Once all the chunks for all the query words are in memory than, we create a list of all the blocks a particular inverted list spans. This is called as Word-Block view i.e. for a word we try to keep all its block within view. We do not un-compress any block yet.
    2. *nextGEQ*
        a. Takes an Inverted List and a Document Id.
        b. Searches the chunk metadata for the right block to un-compress. Once, found only that block is un-compressed using delta and var-byte compression. See Appendix 3.
        c. Returns the next greater than or equal to doc id and its freq.
    3. *closeList*
        a. Takes an Inverted List
        b. Clears all the resources allocated to an inverted list.

  Conjunctive Queries – Document IDs having all the query terms
  Disjunctive Queries – Document IDs with a subset of query terms

- *Ranking of Returned Documents*
  BM-25 score is used to rank the documents.

How to run Index Generator and Query Processor

Complete project is packaged as a Maven Project, so before running anything you have to make sure, you complete below steps.

1. Install Maven Client for Windows/Mac from [5]
2. Make sure, you have Java installed in your machine
3. JAVA_HOME variable should be set as an environment variable
4. MAVEN_HOME variable should be set as an environment variable
5. Make sure maven is set by running '*man mvn'* command

Now, in order to build the project, follow below steps.

1. Navigate to the root directory (Directory with *pom.xml* file)
2. Run '*mvn clean install'.*
3. Make sure the build is succeeded.

Index Generator – Index Generator comes in 3 parts. See below for details on how to run each program and how to interpret the configuration file for each.

1. *Intermediate Postings Generator*
   Description – Generates the intermediate postings from HTML/WET files
   How to run – Run *IntermediatePostingsGenerator.sh* (in root directory)
   Configuration File – *rawpostings.properties* (in root/websearch/intermediate)

Configuration Name	Description
inputpath	Input path for HTML/WET Files
outputpath	Output path for Intermediate Postings
maxtempfiles	Max intermediate files you want; this will decide the size of each intermediate file

2. *Postings Merger*
   Description – Merges the intermediate postings generated in previous step
   How to run – Run *PostingsMerger.sh* (in root directory)
   Configuration File – *merger.conf* (in root directory)

Configuration Name	Description
inputpath	Input path for Intermediate postings
outputpath	Output path for Merged Postings File
filestojoin	Max Files to join at one time
postingsfileformat	File name format for the intermediate postings

3. *Index Creator*
   Description – Creates the final index structures
   How to run – Run *IndexGenerator.sh* (in root directory)
   Configuration File – *index.properties* (in root/websearch/index)

Configuration Name	Description
inputpath	Input path for Merged File
outputpath	Output path for Index Structures
chunkBufferRatio	Total Chunks in the Buffer (Chunk Size=256kb)
writeasbinary	binary/ascii

Query Processor – Query Processor is responsible for answering user queries. See below for details on how to run the program and how to interpret the configuration file.

1. *Query Processor*
   Description – Looks up the Inverted Index structures and answer user queries based on BM-25 score
   How to run – Run *QueryProcessor.sh* (in root directory)
   Configuration File – *processor.properties* (in root/websearch/queryprocessor)

Configuration Name	Description
inputpath	Input path for Inverted Structures
chunkSize	Chunk Size used (Default = 256kb)
cacheSize	Cache Size used (Should me in multiples of 256kb)

Appendix 1 – Inverted List

Document ID and Frequency pair constitute a Word Posting. So if in Document# 2, a particular word has occurred 199 times than (2, 199) is a Word Posting. Collection of all such postings for a word will form an Inverted List for that word.

Appendix 2 – Chunks and Blocks

- Idea behind using Chunks and Blocks is that we want to arrange the postings (See Appendix 1 for Word Postings explanation) of all the words in a way that is efficient enough to retrieve while processing User Query.
- Chunks are basic units for Disk Retrieval and Caching. So we extract Chunks from disk and cache chunks in memory. Chunk Size for this project is 256KB.
- Blocks are smaller units that constitute a Chunk. Blocks have the restriction of exactly 128 Postings per block. Blocks are basic units of Decompression.

- In an Inverted Index, inverted-list for a word can start in any Chunk and Block and can end in any Chunk and Block. Lexicon has the required metadata to figure out the start and end of an Inverted-List.
- Below are the advantages of adopting such a strategy
    - Chunk Size aligns with Disk Memory transfer units.
    - While searching for postings in an inverted list, we can avoid decompressing some blocks since postings within a list are ordered. So rather than decompressing the whole inverted list, we only decompress a very small part of it.

Appendix 3 – Compression Used

Blocks are basic units for compression. This is known as Block wise compression scheme. So whenever we compress/un-compress something, we do it in blocks (precisely 128 postings). There are two levels of compression used in this project. *CompressionDecompressionUtility (Java class)* implements all the below functionality. Below we describe those in detail.

1. *Delta Compression*
   For each posting (only the Document ID, not the Frequency) we take its difference from the previous posting in the block. There is no delta compression for the first posting in the block.
   Also, there is a catch in this which needs to be appropriately handled while decompression. Since, an inverted list of a word can start anywhere within a block, there might be a problem where, while delta compression there is a negative number. This is a problem in *var-byte compression*. So to address this problem, whenever a new list starts, we don't take its difference with the last document id. While decompression, this corner case needs to be handled. Reason for not compressing the first posting in each block is that while processing query, we can skip many blocks. So with this we can process each block independently.
2. *VAR-BYTE Compression*
   VAR-BYTE Compression is the simplest integer compression scheme. According to this scheme if a number is less than 128 than we need 1 byte to store it. If it is less than 128*128 than we need 2 bytes and so on. Advantage of this compression is its speed of compression and decompression. Current version can compress and decompress 50 million integers/s. Though it is even faster than this number.

Appendix 4 – Interface for working with Inverted Lists

Document at a time processing (DAAT) can be approached by following 4 primitives.

1. openList(t) – Opens an inverted list given a word

2. nextGEQ (list, k) – Gets the next posting bigger or equal to k
3. getFreq(list) – Get the frequency associated with current doc id
4. closeList(list) – Closes the list i.e. frees up resources allocated

References

[1] http://commoncrawl.org/2014/04/navigating-the-warc-file-format/
[2] http://engineering.nyu.edu/~suel/cs6913/javaparser/
[3] https://github.com/commoncrawl/example-warc-java
[4] http://commoncrawl.org/2016/10/september-2016-crawl-archive-now-available/
[5] https://maven.apache.org/download.cgi