**Query Parser**

**1. Introduction:**

The whole project consists of 4 parts: indexing, merging, compressing, and querying.

Indexing module first parses the data set NZ, and then generates temporary inverted index file of each pair of index and data.

Merging module merges all temporary inverted index file into one final inverted index file using external sorting.

Compressing module compresses the final inverted index using variable byte coding.

Querying module handles the query items and then returns Top 10 results in BM25 Ranking.

**2. Modules:**

**2.1 Indexing module:**

(1) Inverted index

An **inverted index** (also referred to as postings file or inverted file) is an index data structure storing a mapping from content, such as words or numbers, to its locations in a database file, or in a document or a set of documents.

There are 2 forms:

Word/WordID🡺 <DocID1, TF1> <DocID2, TF2> ……< DocIDi, TFi>…… <DocIDn, TFn>

Word/WordID🡺 <DocID1, TF1 , {POS1,POS2,…}> <DocID2, TF2, {POS1,POS2,…}> …… <DocIDn, TFn, {POS1,POS2,…}>

* *DocID is assigned when parser parsing the page and stored in chronological order*
* *WordID is assigned when parser parsing the page. Using treemap, duplicates can be removed, and only new word gets new ID.*
* *TF(Term Frequency) means the number of the word which occurs in corresponding DocID.*
* *POS means the position of the word in the Doc’s content.*

According to the 2nd case, you can accomplish the tasks of snippets and adjacent words ranking. In this project, I just make use of the 1st case. This project will not focus on them.

(2) URL map

{ URLID, URL, lenOfDoc/NumOfWords}

* *DocID is assigned when parser parsing the page and stored in chronological order*
* *URL corresponds to its DocID.*
* *NumOfWords means how many words are in the Doc, which is required in BM25. In my project, I just use lenOfDoc instead of NumOfWords. Assuming that the average length of words is nearly constant, lenOfDoc can do similar function and make sense.*

*It can simplify my task.*

(3) Lexicon

There are 3 forms:

{ WordID, Word, FT, maxDocID, Length, Startpoint} // for general Inverted index

* *WordID is assigned when parser parsing the page and stored in chronological order*
* *Word in charactors.*
* *FT means the number of Docs which contains this term/word.*
* *maxDocID means maximum DocID in this word’s posting list.*
* *Length means how many chars the word takes in the inverted index which is in the disk.*
* *Startpoint is the sum of Lengths in previous words.*

Finally, the information can be loaded from the Lexicon file into LexiconHashMap(unordered\_map), which can help me find info very quickly.

{ WordID, Word, FT, startDocID, maxDocID, length, startpoint} // for D-Gap Inverted index

* *WordID is assigned when parser parsing the page and stored in chronological order*
* *Word in charactors.*
* *FT means the number of Docs which contains this term/word.*
* *startDocID/miniDocID means minimum DocID in this word’s posting list.*
* *maxDocID means maximum DocID in this word’s posting list.*
* *Length means how many chars the word takes in the inverted index which is in the disk.*
* *Startpoint is the sum of Lengths in previous words.*

Using D-Gap, only the 1st DocID(startDocID/miniDocID) in this word’s posting list is absolute/real DocID, and the rest are DocID difference.

{ WordID, Word, FT, < startDocID1, chunkMaxDocID1, len1>, …… ,< startDocIDk, chunkMaxDocIDk, lenk>, maxDocID, length, startpoint } // for chunkwised D-Gap Vbyte Inverted index, say at most 128 postings a chunk

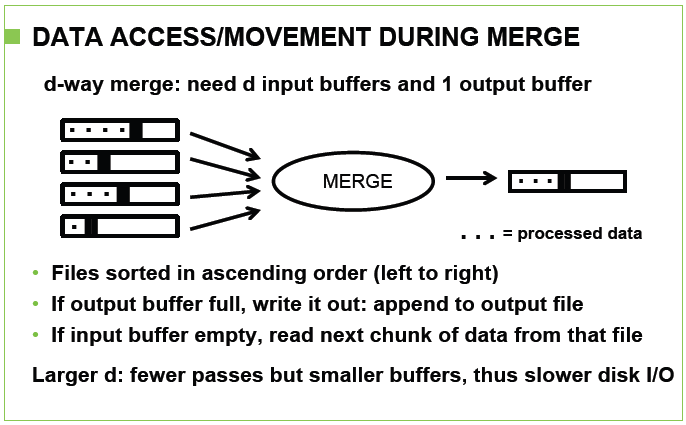
* *WordID is assigned when parser parsing the page and stored in chronological order*
* *Word in charactors.*
* *FT means the number of Docs which contains this term/word.*
* *startDocIDi means minimum DocIDi in this word’s posting CHUNKi.*
* *chunkMaxDocIDi means minimum DocIDi in this word’s posting CHUNKi.*
* *maxDocID means maximum DocID in this word’s posting list.*
* *leni means how many chars the word takes in the posting CHUNKi.*
* *Startpoint is the sum of Lengths in previous words.*

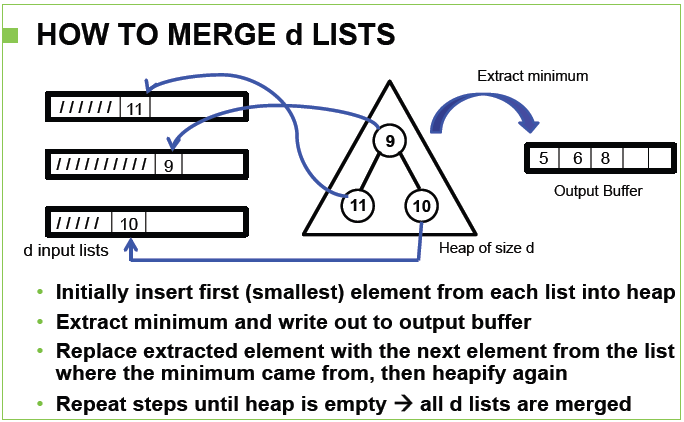
In chunkwise variable bytes coding compression way, if expected DocID is bigger than chunkMaxDocIDi , go forward until expected DocID is smaller than or equal to chunkMaxDocIDk. After decompressing CHUNKi, check if it exits in this posting chunk.

In this project, I use the first 2 forms, general variable bytes coding compressed inverted index and D-Gap variable bytes coding compressed inverted index.

**2.2 Merging module:**

Here I modified the provided external sorting (IO-Efficient Sorting) code.





The detailed information of IO-Efficient Sorting can be referenced in the slides, so it can be skipped.

In the temporary inverted index, I don’t use any compression coding algorithm to handle it.

I create a ***struct posting*** as an element,

***typedef struct {***

***unsigned int WordID;***

***unsigned int DocID;***

***unsigned int TF;***

***} posting;***

and re-define KEY(z),

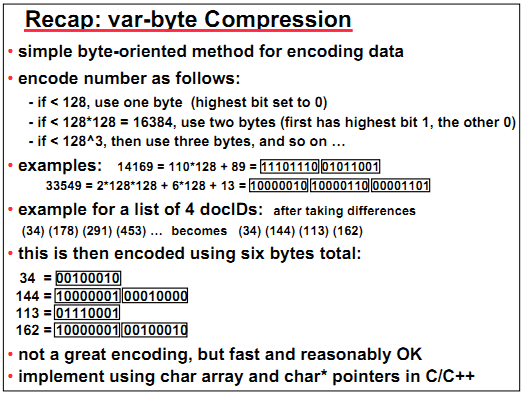
*#define KEY(z) (\*(posting \*)(&(heap.cache[heap.arr[(z)]\*recSize])))*

In the data structure HEAP, if children’s WordID is smaller than parent’s WordID, or children’s WordID is equal to parent’s WordID but children’s DocID is smaller than parent’s DocID, exchange them.

Finally, it comes out a file of uncompressed global inverted index.

**2.3 Compressing module:**

Here I use variable bytes coding to do compression and decompression of inverted index. The detailed information of var-byte compression/variable byte coding can be referenced in the slides, so it can be skipped.

. 

When compressing the inverted index, 2 auxiliary global data structures are created:

*vector< unsigned int > FT; // the number of documents that contain term t*

*vector< pair<string, vector< pair<int,int> > > > Lex;*

*// vector subtitile is worded. <word in charactors, vector <maxdocid,len>>*

Using these data structures, Lexicon table with more information can be generated.

**2.4 Querying module:**

(1) Pre-loaded URL HashMap and Lexicon HashMap(unordered\_map)

At first, it takes some time to build a Lexicon HashMap and a URL HashMAP. Lexicon HashMap can help find WordID very quickly and target the lineNum of this word.

*unordered\_map < int, string > urlHashMap; // < URLID, url >*

*unordered\_map < string, pair<int,int> > lexHashMap; // < word ,pair<wordid,maxDocID> >*

(2) Document-At-A-Time Query Processing

Here Document-At-A-Time Query Processing will be implemented.

Properties of DAAT: Document-At-A-Time Query Processing is faster and better than Term-At-A-Time; and hides underlying implementation of inverted lists: disk layout and index compression.

Using conjunctive semantics, it can easily find the intersection set and their ranks.

Interfaces: openLIST(t),closeLIST(lp),nextGEQ(lp,k),getFreq(lp)

The detailed information of DAAT can be referenced in the slides, so it can be skipped.

(3) BM25 Ranking Function

There are about many ways to evaluate a page’s rank, for example, PageRank, HITS, Topic Sensitive, Log-Based methods.

In information retrieval, Okapi BM25 is a ranking function used by search engines to rank matching documents according to their relevance to a given search query.

* *N: total number of documents in the collection.*
* *ft : number of documents that contain term t.*
* *fd,t: frequency of term t in the document d.*
* *|d|: length of document d.*
* *|d|avg: the average length of documents in the collection.*
* *k1 and b: constants, usually k1 = 1.2 and b=0.75.*

Here I use AND/conjunctive semantics to find intersection set, pages which contain all terms in the query.

(3) Method to get Top10 pages

Here I create a ***Priority Queue*** of size 10 and made of ***Min-HEAP***, which can help me pop Page Node, Rank of which is smallest. At the end, a vector of Page Nodes stores top 10 results in ***ascending order*** of Rank. If this vector **is *read from ending to beginning***, top 10 results in ***descending order*** can be generated soon.

**3. Implementation and Performance:**

**3.1 Explanation:**

At the beginning, I did my coding in VMware and 32-bit Ubuntu 10.04 platform. However, my virtual machine only has only total 20GB space. I have no enough remaining space for handling total NZ data set. Also, 32-bit platform may not allow me to hold a file which is larger than 2GB. My uncompressed final inverted index file is larger than 2GB, which make me some trouble.

Thus, I use 730 out of 4180 files to start my work then. The total size of uncompressed final inverted index file is about *1.9 GB (2032142196 bytes)*. The format is contiguous triples of

<WordID,DocID,TF>.

Final compressed inverted index file using ***Variable Byte Encoding*** is about *640.0 MB (671126090 bytes)*. Corresponding Lexicon and URL files are *39.0 MB (40925838 bytes)* and *32.4 MB (33997107 bytes)* respectively.

Final compressed inverted index file using ***Variable Byte Encoding*** and ***D-Gap*** is about *340.5 MB (357056236 bytes)*. Corresponding Lexicon and URL files are *46.4 MB (48616563 bytes)* and *32.4 MB (33997107 bytes)* respectively.

Final compressed inverted index file using ***Variable Byte Encoding*** and ***D-Gap*** only takes 18% space of unprocessed uncompressed final inverted index file.

**3.2 Exception in data set:**

Also, I find some problems and exceptions in NZ data set and temporary inverted index file.

1. 2406\_data cannot be decompressed. It’s a bad file!

2. 959\_data is full of 404 Not Found. Therefore, generated inverted index file is empty. If someone wants to use IO efficient algorithm provided in the course website, he/she should take this case into consideration and get rid of it. Otherwise, the program will end up with this exception. In addition, several data files can be combined as a chunk to be processed.

**3.3 Performance of indexing:**

On the platform of 32-bit Ubuntu10.04 VMware virtual machine with 2GB memory, it costs 1595s to make temporary inverted index files using 0\_data to 729\_data.

**3.4 Performance of merging:**

On the platform of 32-bit Ubuntu10.04 VMware virtual machine with 2GB memory, it costs 80s to make final inverted index file by merging these 730 temporary inverted index files. I have tried 2-passes way to do external merging but it takes longer time.

**3.5 Performance of compressing:**

On the platform of 32-bit Ubuntu10.04 VMware virtual machine with 2GB memory, it costs 58430ms to make general Variable Byte compressed inverted index file. In D-Gap Variable Byte compression way, it only takes 42420ms.

**3.6 Performance of querying:**

Finally, I move querying module into my dual system Ubuntu 12.04 LTS.(AMD Phenom(tm) II P650 Dual-Core Processor 2.60 GHz, 4.00GB)

The speed is fairly fast. It usually takes 10-100ms to search a single word query. In addition, it takes several hundred milliseconds to search the query of multiple words, which are frequent in both in *ft* and *fd,t*. If intersection set is very small, it just takes several ten milliseconds.

For example, a query consisting of terms which are frequent both in *ft* and *fd,t*, “to be or not to be”. It only takes 800ms to finish querying.

**4. Limitations:**

(1) I have a design and data structure for chunkwise compression inverted index. However, I have no more time to finish it.

(2) Cache can be much helpful in my querying.

(3) I should predict the size of temporary inverted index file first. A 64-bit platform is suitable to hold a larger file.

(4) If I compressed temporary files using good compression algorithm in intermediate, 32-bit platform may not be the bottleneck.

(5) Though I implemented D-Gap and Variable Byte Encoding, I find the query speed is slower than Variable Byte Encoding without D-Gap. It’s probably my problem of data structure designing.

(5) Snippet should be included.

(6) Web accessible interface should be included.

**5. References:**

Wikipedia, course PPT slides.