Hengning Zhang hz1704

My implementation can be described as three components:

1. Parsing the .trec file into an intermediate posting file.
2. Sort the intermediate postings file with unix sort.
3. Read through the sorted intermediate postings and build the inverted index.

Explanation for each part:

1. Parsing the .trec file into an intermediate posting file:

I wrote my own parsing method. The idea is to have a flag that is set to be true whenever <TEXT> is encountered and it is set to false when </TEXT> is seen. Read everything in between and take those that are only composed of A-Za-z0-9. I originally was doing this using regex but I thought it is probably better to check on the fly because sometimes the edge of the buffer is not a whole word. So finally I decided to read the char array one character at a time.

Possible improvements: change this to using unicode to be able to deal with more character.

Load 1 GB of the .trec file into the buffer. While inside a document, I would take any term that is in between a blank space and a '\n'. (Some terms need to be divided into smaller terms because there are symbols in them.) and push the terms into a vector. After finishing reading a document, write the document id and url into the document map buffer, in the form of "docID url\n". Sort the words in the vector alphabetically, then do a counting of their frequencies in this document. Output it in the form of "term docID frequency\n" into the intermediate postings buffer. Clear the vector to proceed reading the next document.

Do this until the .trec file is exhausted.

Whenever a buffer reaches its set size limit, dump the content into their corresponding file.

This step takes around 42 minutes to finish. Creating an intermediate posting file that is 20.8 GB.
The document map takes 228 MB.

2. Unix sort:

```
sort -S 50% --parallel=4 -k1,1 -s intermediate_postings.txt >
sorted_postings.txt
```

Use this command to sort them by the first column, which is the term, and keep the other parts stable.

For the same term, the one with smaller docID will always be generated first in the intermediate postings so stable sorting would keep it there.
Sorting takes around 40 minutes. Size of the file stays the same.

3. Index building

Since the intermediate posting is a processed document, its format is known.
All lines are in the form of “term docID frequency”.

Compression done using varbyte provided in the slides.

Load char array from .trec file into an 1 GB input buffer.
We also have an unsigned char output buffer of size 1000000 for the compressed numbers. It is kind of weird that when I was trying to create buffer sizes that are bigger it did not work…

We are doing a similar parsing here, reading chars one by one and forming words (term/docID/frequency) by using blank space and ‘\n’ as dividers.
We have two vectors for each term: one for docIDs and one for frequencies, they should always have the same size.
Push the docID and frequencies of the term into vectors while reading.

The protocol/form I did for the inverted index is

Number of postings + last docID in this block + compressed docID block size + compressed frequency block size + docID block of 64 + frequency block of 64

Does not allow galloping but at least we do not need to compress everything in a block when reading. Reading one varbyte tells us if the one we are looking for can be in this following block and reading 3 varbytes tells us how far to skip ahead.

Here’s how it is implemented:
We have 3 buffers here:
Output buffer: the one mentioned above, to write into the real inverted index file
temp_docID: holds at most 64 compressed docIDs
Temp_frequencies: holds at most 64 compressed frequencies

When reaching a different word, deal with the docID and frequency vectors:

1) Write the term and current location of file into lexicon

2) Write the compressed docID_vector.size() into output buffer.

3) Looping through the docID vector and frequency vector:
   Write a compressed docID into temp_docID
   Write a compressed frequency into temp_frequency

Maintain a counter to show if we are reaching 64
If we reach 64, a block should be built.

4) If compressed blocks' sizes+8 bytes+current buffer size>buffer size limit, write the current buffer content into file. Because the buffer is not dynamic. Adding 8 bytes is because the size of the temp_docID and temp_frequency can take up to 4 byte for each number (2^21 only encodes up to 2097152 and as we have more than 3 million documents the docIDs may take 4 bytes each). That would make 64 of them taking up to 256 bytes. Encoding 256 takes 2 bytes in varbyte so encoding temp_docID and temp_frequency sizes takes up to 4 bytes.
   The other thing we want to encode before the actual block sizes is the last docID in this block, which also takes up to 4 bytes. Thus we need to check if all of this fits.

5) After dealing with the size limit (or not if we don't reach it):
   Write the last docID we just compressed into output buffer (This is the last docID for this block)
   Write the size of temp_docID and temp_frequency
   Write the actual blocks of data

6) Do all of the above until we reach the end of the sorted intermediate posting file

Then finishing up with copying all the huge chunks of code above for another 3 times because we want to deal with:

a. The last block for each term that is not necessarily consisting 64 docIDs
b. The last term's information because there will not be a next term to trigger the writing
c. The last block that may have size<64 for the last term

Feels messy but I think they are essential.

Wohoo! We are done with creating the inverted index file!
It takes 750 seconds to build the inverted index from sorted intermediate postings.
lexicon.txt takes up 399 MB.
Inverted index takes up 5.35 GB.

File retrieval system has not been made yet.
But I tested that it actually can get me the information I want by randomly looking up a term in the sorted intermediate posting and searching for the information with lexicon and inverted index.
See testing_sampleing.txt for the actual postings and testing_result.txt for the retrieved values.

To compile and run the files:
Make sure you have g++ installed and there should be a "fulldocs-new.trec" in the same folder.
g++ -o parsing parsing.cpp

./parsing
This will read from that file and result in a “intermediate_postings.txt” and a “docMap.txt”. There will be some console logs to show how many files are processed and how much time elapsed.

Then call unix sort to sort the file.

```
sort -S 50% --parallel=4 -k1,1 -s intermediate_postings.txt > sorted_postings.txt
```

Requires half of your computer’s memory! Make sure to **close Google Chrome**.
Creates “sorted_postings.txt”, name is self-explanatory.

Then the last step is to build the index!
g++ -o building_index building_index.cpp
./building_index

To try testing yourself:
Think of a term you like to test my code on.
Change QUERY_TERM in sampler.cpp and tester.cpp to it.

g++ sampler sampler.cpp
./sampler
First 200 appearances of sample would be printed in console.

g++ tester tester.cpp
./tester
First several blocks of info about that term you chose will be printed in console.
I would recommend something that starts with “a”, for sampler.cpp is searching in sorted_postings.txt for that term with ifstream.