Better Inverted Index Compression by DocID Reassignment

Hengning Zhang hz1704

Background:
Traditional web search engines are based on inverted indices. An inverted index allows fast queries in a given document collection. Being able to compress the inverted index further allows more caching in main memory, which would in turn greatly improve the query performance. This would also make the inverted index based web search engine much more scalable since real life document collection can be of astronomical size.

Inverted index:
A file that records which terms exist in which documents.
Traditionally the inverted index would be stored in the form of document ids + frequencies.

Compression methods:
Variable byte(VarByte) encoding, Gamma coding and Binary interpolative coding are used in this project.

VarByte is using an indicator bit at the beginning of each compressed byte to indicate whether this is the last byte of an integer, in order to let the decoder decode the compressed file without ambiguity. Other than that it is similar to 32 bit integers.

Gamma coding does pretty well in compressing small numbers. It is basically encoding an integer n>=1 using a unary representation of 1+(logn) to represent the length of the logn lower bits of n’s binary form. Using 1+2logn bits in total to represent a small number does not work very well if the number is big.

Interpolative coding does really well in compressing clustered data. It compresses the document IDs directly without turning them into gaps. The idea is that given the smallest and largest item of a monotonically increasing array of integers (no duplicates) and the array’s length, we can encode the middle integer d[m] by d[m]-l-m with log2[l-r-n+1] bits where n=length of array, m=n/2, d is the array, l is the min value and r is the max value. Note that if l-r=n, we actually do not even need to store the value because every integer between l and r exists in the array. [2]

Gamma coding and interpolative coding were tested on reassignment by GOV2’s forward index sorted by url.

Since Binary Interpolative coding capitalizes on closely grouped document ids the most, we shall test the document reassignment result mainly by Binary Interpolative Coding (BIC/IPC)

Parsing the GOV2 collection:
I implemented the parser myself. Zlib library was installed to do this job and it cost me a lot of time and effort because of compatibility.

Go through each subdirectory of the 273 folders in GOV2, decompress each file in there and read into a buffer.
Read the tag's content when encountering a "<". This is to let the program know which section it is reading.
When encountering <DOCHDR> we encounter the header section of a document. The url and docID is taken note of at each dochdr.
<HTML>/<html> indicate the start of sections we need to parse out the words. Parse out every continuous string consisting of English characters+numbers.
</HTML>/</html> are the end of the important section and the program stops the parsing and waits for the next html section.
When parsing, the program takes notes of how many distinct words and the size of the document.
When getting to the end of a document, </DOC>, write down the notes taken into a file that holds the initial document id, url, distinct word count and size of each document. This file is to be used in the document id reassignment process. Call this file the docInfo file.
Also, write term+docID+freq into a buffer for each term appearing in this document.
When we finish running the process we get a file that has all the intermediate postings but unsorted.

Reassign document id:
Used the gathered DocInfo, call unix sort to sort by url and by distinct word count.
Create an array A of size max document id in the collection.
Read through the sorted docInfo file, and hold a counter i.

```
For each docID read
    assign A[docID]=i
    i++
```

Then read through the unsorted intermediate postings file and map each of the old docIDs to the new ones.

Building the Inverted index:

Document collection is parsed at first. In this report we are using GOV2 dataset, a crawl of the .gov web from 2004. It consists of 25 million web documents with about 500GB size.

The first step is parsing GOV2 and extracting all the terms between spaces and punctuation marks, and writing them in the form of “term docID frequency” form. Within each document ID the terms are sorted. This is the forward index that we will use later as a bipartite graph input for the document id reassignment algorithm. The bipartite graph has terms on one side and document ids on the other. Each edge represents the existence of a term in a document.

During parsing, we also create a document info file that contains each document’s document id and their corresponding url. This file will also be used in document reassignment.

Based on the former work of Silvestri [1], sorting the forward index by url (alphabetically) can already achieve great compression improvement. After sorting the document info file by the url column, simply create an array with the size of MAXDocID, read through the sorted document info file, and map each old docID to the new ones.

Experiments and Discussion for the sorting approach:
Use the array to map each entry’s docID to the new one in the forward index, then sort the forward index by term and docID with unix sort.

The forward index is approximately 80GB in size.

Reading through the processed forward index we can create the inverted index with IPC compression. A lexicon file is also created to use the inverted index.

The protocol for compressing inverted index with IPC is as follows:
Use VarByte to encode the total postings.
For each block, use VarByte to encode each block’s docID count, starting docID and ending docID so that we can use IPC to decode.
Use IPC to encode the 64-docID block with the docID count, starting docID and ending docID.
Since this encoding is bit-by-bit, we first encode in a vector<bool> and use that vector<bool> to gather all the partial bytes into whole bytes. In case there are leftover bits in the last byte, we fill the vector<bool> to a size of multiple of 8 so that the last byte is a whole byte.
Then use gamma coding for the frequencies, 64 in a block.

The protocol for compressing inverted index with gamma coding only is slightly different. Gamma coding is compressing the gaps instead of docIDs themselves.

Here are the results:
Using the forward index sorted by Url: 7295 seconds 3.36gb.
Using the forward index sorted by distinct word count: 7684 seconds 4.08gb.

It is reasonable to use url sorting because we can sort of expect websites with similar urls to be about the similar topic, in turn, containing a similar set of words. However there is a hard constraint on this kind of sorting method: every document must have a url-like identifier to allow sorting by url. In some other context, for example, wikipedia, this sorting method may not even be useful anymore since every url is so similar and there are so many terms that are close in the english alphabet but mean totally different things. The reason why this is a valid reassignment method is that the implementation is very straightforward and is very quick to execute. It can give a reasonably good result at a low cost.

There is definitely a larger chance of having more similar words in two documents containing more distinct words. However there is no heuristic of why the gap should be smaller than random assignment of document IDs. Compared to sorting by url, it is not that effective in compression rate using the GOV2 dataset. However, this method does not require any url-like identifiers for the documents, making it more universal than the sorting by url method. Overall, it is a similar approach as the sorting by url, providing a slightly worse compression rate but has no constraint at all.

Other Relevant Previous work on other non-trivial docID reassignment:
Prior solutions focused on dense-graph-like approaches.
The document collection can be seen as a dense graph where documents are vertices and edges represent connection between documents in terms of similarity. Edge weights represent the similarity extent, for example the number of common terms in these documents.
These proposed algorithms try to use graph traversal techniques such as creating a maximum spanning tree and traverse the tree with DFS and reassigning the docIDs with the order they are encountered in DFS. Or The Traveling Salesman approach, which looks for a way to traverse every node in the graph and maximize the total number of cost (sum of edge weights) on the route.

Then there are optimizations of these graph algorithms by making the graphs sparse, through hashing and selecting significant connections between the vertices. [5]

BP algorithm[4,5]:

```
Algorithm 1: Graph Reordering via Recursive Graph Bisection
Function RecursiveBisection($T, D, d$)
    In: Bipartite graph $(T, D)$ with $|D| = n$ vertices and recursion depth $d$
    $D_1, D_2 =$ SortAndSplitGraph($D$)
    for iter $= 0$ to MaxIter do
        forall $v$ in $D_1$ and $u$ in $D_2$ do
            $gains_{D_1}[v] =$ ComputeMoveGain($T, v, D_1, D_2$)
            $gains_{D_2}[u] =$ ComputeMoveGain($T, u, D_2, D_1$)
        SortDecreasing($\text{gains}_{D_1}$, $\text{gains}_{D_2}$)
        forall $v$ in $gains_{D_1}$ and $u$ in $gains_{D_2}$ do
            if $\text{gains}_{D_1}[v] + \text{gains}_{D_2}[u] > 0$ then
                SwapNodes($v, u$)
        if No Swaps Occurred then
            $iter = MaxIter$
    if $d <$ MaxDepth then
        $D_1 =$ RecursiveBisection($T, D_1, \lfloor n/2 \rfloor, d + 1$)
        $D_2 =$ RecursiveBisection($T, D_2, \lceil n/2 \rceil, d + 1$)
    return Concat($D_1, D_2$)

Function ComputeMoveGain($T, v, D_a, D_b$)
    In: Bipartite Graphs $(T, D_a), (T, D_b)$ with $|D_a| = n_a$, $|D_b| = n_b$ and $v \in D_a$
    $gain = 0$
    forall $t$ in $T$ do
        if $t$ connected to $v$ then
            $d_a, d_b =$ number of edges in from $t$ to $D_a$ and $D_b$
            $gain = gain + d_a \log_2(\frac{n_a}{d_a+1}) + d_b \log_2(\frac{n_b}{d_b+1})$
            $gain = gain - (d_a - 1) \log_2(\frac{n_a}{d_a}) + (d_b + 1) \log_2(\frac{n_b}{d_b+2})$
    return gain
```

Pseudocode from [5].
Compression of inverted index by recursive graph bisection is the state-of-the-art algorithm, implementation detail will be discussed in the following section.

Implementation of BP algorithm:

The sorted intermediate postings file created by reassigning document id through sorting by url is used to create a forward index to represent the bipartite graph of term->document.
The implementation of this is straightforward: read through the intermediate postings file and gather all docID for one term. If #count>some threshold we set, write them in the form of term+(a bunch of docID gaps) in the forward index file.

The threshold I chose was filtering out terms that appear in over 8192 documents.
I tried 4096, 8192 and 16384 with the idea that decreasing the number of “important terms” might give me a boost in performance in terms of running time.
However, the filtered words’ intermediate posting file’s size does not seem to change much. Although the number of terms decreased to 20000 from 40000, the forward index still takes around 8GB in string form. So I stuck with 8192.

Putting all of these docIDs in the bipartite graph in the form of adjacency lists in memory takes around 13 GBs. This is not very scalable but I did not come up with a better solution to reduce the memory usage. Since I am running a 64 GB memory machine this is still relatively runnable in my machine.
Call this adjacency list array termEdges. It contains an adjacency list for every significant term.

How to handle bisection and swapping:
Array to store original docID->new docID so that docIDs can be updated while swapping
Store this in memory would be fine since it is only 25 million ints which is about 100 megabytes. This should be pretty easy for DIMDS to handle.
Always use the array that contains 25 million docIDs, only swap there.

Pass start and end for bisection instead of bisections themselves.
When done, just read through this array and the inverse of it is how to assign the new docIDs.

Precompute number of edges to both of the bisections:

```
Create a mask buffer M of size max document id with all 0.
For each docID d in one of the bisection:
    Assign M[d]=1;
For each docID d in the other bisection:
    Assign M[d]=2;
Create an array of pairs to store the precomputed edge counts.
For each term t in termEdges:
    pair<int,int> edges
    For each docID in adjacency list of t:
        If M[docID]==1:
            Pair.first++;
        If M[docID]==2:
            Pair.second++;
```

Read through the forward index, aggregate the counts for each term and get each term’s number of edges to each bisection.

This would cost 25m+#termedges to get edge counts for each term.

Calculate the move gain with this information:
Move gain is to determine whether moving a docID to the other bisection is making the assignment a more clustered one.
moveGain() takes a docID d, the two bisections, and the array of edge count we created in the previous step.
With a given document id docID:

```
For every term t in termEdges:
    If Binary search(docID, adjacency list of t):
        Calculate the Gain
```

This is doing binary search for each term every time we want to calculate the move gain for a document.
I did not have enough time to run a complete BP algorithm on GOV2 because the time it takes to run iterations already takes too long to run.
I ran 3 iterations on the highest level:
Binary search 14847s for first iteration, 9000s for second iteration and 7000s for the third.
Reassigning using the result from the third iteration gives a compression of 4.75 GB with Interpolative coding.
The compression rate should increase after each iteration and the result I got is limited because I only ran 3 iterations where 7 million swaps happened.

Possible improvement:
Instead of running binary search on every adjacency list of termEdges, we can use **fractional cascading** [6] to reduce the binary search time, since all the adjacency lists consist of monotonically increasing integers without duplicates and they have a common constant max length. And the most important of all, we are searching for the same document ID in all of them.

The implementation of Binary interpolative coding is referencing the pseudocode in [2].
BP algorithm based on pseudocode in [5].
Zlib library is included to

Machine specification (the one i am using):
AMD Ryzen 2600X
64GB Memory
Windows 10 Home (Chinese Simplified)

How to run the code:
Make sure to have Zlib installed and linked to g++.
Make sure to have the GOV2 downloaded as a folder with 273 folders.
I sort of hard coded the file opening by traversing GOV2/GX000 through GOV2/GX273 and from 00.gz to 99.gz to exhaust the gz files in the folders.

```
g++ -o decompress_and_parse decompress_and_parse.cpp -lz
```
This is to decompress the GOV2 and parse it to postings.
-lz is required to use zlib or mingw compiler claims to not be able to find the zlib library.

```
sort -S 80% --parallel=4 -k2,2 source.txt > target.txt
```
Use unix sort like this to sort the docInfo by url.

```
sort -S 80% --parallel=4 -k4,4n source.txt > target.txt
```
Or like this to sort by distinct count, note the n is required to sort by numerical number instead of string. So 1<2<10 instead of 1<10<2.

Compile reassign_docID_url.cpp to reassign the documents with the sorted docInfo by distinct count.
Compile reassign_docID_distinct_count.cpp to reassign the documents with the sorted docInfo by distinct count.

Sort the intermediate posting file by term and then by docID.
```
sort -S 80% --parallel=4 -k1,1 -k2,2n source.txt > target.txt
```

Compile index_building_gamma.cpp to use gamma encoding.
Compile index_building_ipc.cpp to use ipc building.

You need to have the sorted intermediate posting file generated after reassigning document ids by sorting by url to run BP.

Compile and run termFilter.cpp to filter out terms that appear in more than 8192 documents.

Compile the
To run an iteration of BP algorithm:
Create a .txt file with 0-25205178 each on one line.
Change the “reassignment” string variable in recursiveBisection.cpp to the .txt file name you just created.

Designate an output file to store the reassignment after this iteration.
Compile recursiveBisection.cpp and execute it.

Compile and run reassign_bp.cpp to do the reassignment on the sorted intermediate posting file generated after reassigning document ids by sorting by url.
Compile and run index_building_ipc.cpp to build the final product of this project.

References

[1] Silvestri, Fabrizio. “Sorting Out the Document Identifier Assignment Problem.” *ECIR* (2007).
[2] Pibiri, Giulio Ermanno. “On Implementing the Binary Interpolative Coding Algorithm.” (2019).
[3] Shuai Ding, Josh Attenberg, and Torsten Suel. 2010. Scalable techniques for document identifier assignment in inverted indexes. (WWW '10).
[4] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. 2016. Compressing Graphs and Indexes with Recursive Graph Bisection. (KDD '16).
[5] Mackenzie, Joel et al. “Compressing Inverted Indexes with Recursive Graph Bisection: A Reproducibility Study.” *ECIR* (2019).
[6] Chazelle, B., Guibas, L.J. Fractional cascading: I. A data structuring technique. *Algorithmica* 1, 133–162 (1986).