Information retrieval - Project proposal
SPIMI Index Construction

Addis Gebremichael, Manuel Gerardo Orellana

1 Introduction

This project aims to demonstrate the optimal use of Single-pass in-memory (SPIMI) indexing algorithm by implementing it for information retrieval of textual data. An alternative index construction algorithm, i.e. Blocked sort-based indexing, has excellent scaling properties, but it needs a data structure for mapping terms to termIDs which can be memory inefficient for very large data collections. Hence, a more scalable option is the SPIMI index construction algorithm which uses terms instead of termIDs, writes each block's dictionary to disk, and then starts a new dictionary for the next block. Thus, it can index collections of any size given there is adequate disk space. The following sections discuss implementation techniques applied to meet the deliverables with in the scope of the proposed project, which mainly include index construction, boolean search and the results obtained during experimental testing.

2 Inverted Index Construction

The construction of the SPIMI inverted index procedure starts from analyzing text into tokens and keeping the transformed documents on disk. Next, the initial inverted index is built based on the transformed documents. Since the dictionary, used to index the documents in which each term appears, grows in size and occupies the entire memory, the algorithm flushes the dictionary block onto disk.The merge process then merges these individual blocks to create the final index. Furthermore, since the document ids generated while building the index simulate a large dataset collection, compressing the posting list can bring visible efficiency.
The following sections elaborate the implementation techniques used for the above mentioned procedure.

2.1 Document Text Analysis

The pre-processing of terms into tokens consists of a sequence of transformations that has been applied on the raw datasets, such as whitespace

tokenization, case normalization (to lowercase), elimination of stop-words, and stemming. Thus, for tokenizing and stemming we used an external library, i.e. Lucene. Moreover, to process case normalization and stop-words we defined a simple code in the project.
Basic information leading to the location of the transformed data collection, the generated document Id's and name of the document is written to disk in json format which can also serve as a mapping between the actual transformed document and document Id's. The document Id's we put in place has an Integer data type and starts from a large number incrementing by 1 to simulate real case scenario of the size of a posting list during index construction.

2.2 Single-pass in-memory Indexing (SPIMI)

During the SPIMI index construction, the data type chosen for implementing the posting list is a user-define custom ArrayList which has the feature to dynamically grow in size with respect to the increasing document Ids for a particular term. The custom arrayList starts from a capacity of one and doubles it's size when arrayList becomes full in order to minimize wasted memory allocation. In addtion, as a dictionary to map term with document Ids, we used a TreeMap which provides an efficient means of storing key/value pairs in key based sorted order, and allows rapid retrieval. Hence, the SPIMI algorithm does not need to sort the terms in the dictionary when the memory is full and is about to flush to disk. The treemap guarantees that its elements will be sorted in ascending key (term) order at the cost of computational time rather than memory costs being incurred.
Essentially, the SPIMI algorithm indexes documents from a batch as long as there is space on disk and that the dictionary can fit in memory. However, since we are implementing this algorithm in Java and java doesn't provide suitable control over memory management, as discussed last time, we are building a simulation of the algorithm particularly when constructing index block by block as a result of memory being full. The memory estimation technique applied is discussed next.

2.2.1 Memory Size Estimation

The memory estimation techniques employed consists of estimating the dictionary memory consumption to simulate what the size of the dictionary is when memory can be full. During the construction of the index, the algorithm has to estimate the dictionary size in memory and once it's full it also does another disk block memory estimation so as to fit the dictionary block on disk. Hence, estimating the dictionary size in memory for a single entry of term-to-posting list puts into consideration the following.

- References to an object (including empty ones) in the posting lists:

4 bytes *times* capacity of the posting list. Note that Capacity is the length of the ArrayList while Size on the other hand is the number of elements in the ArrayList.

- Document id represented by Int primitives wrapped to an Integer: 4 bytes per primitive wrapped in an object with 8 bytes consumed by headers in our 64bit platform with CompressedOops and 4 bytes lost to alignment. Thus, this adds up to the size of a posting list *times* 16 bytes. Note that the document Ids are represented by a variable byte when the posting lists are compressed.

- Memory consumption of the posting list array itself: 4, 8 and 4 bytes for reference, header and alignment lost to the custom ArrayList itself. In addition, it also considers the memory consumption of two integers in the custom ArrayList that are used for maintaining the capacity and it's size.

- Memory allocation for a term: 2 bytes per character *times* the size of the String representing the term *plus* cost of reference to the String object.

- Memory allocation for the dictionary Tree map object is assumed to be a negligible amount and does not put much influence, hence, it is ignored.

In the above stated manner, the SPIMI index builder has a reasonable estimation of memory consumption for the growing dictionary TreeMap and when memory is full it initiates an estimation of the dictionary's size on disk blocks before flushing out to disk. Since our development environment, i.e. Windows, uses Unicode as a character-encoding standard, it allocates one byte/character. Thus, with this premise and the assumption we took for one disk block being 64 KB, we first Stringify the dictionary and divide it into blocks that fit into the allocated disk block space.

2.2.2 Binary Merge

The SPIMI index blocks on disk need to be merged so as to obtain a final index which has a single entry for one particular term. Hence, since each block's postings lists were written in sorted order, merging blocks can be accomplished by a simple linear scan through each block. Moreover, with binary merge, a single block is scanned efficiently with a logarithmic complexity as opposed to other techniques which scan the first block *n-1* number of times, where *n* is the total number of blocks before merging.
The merge logic employed behind the algorithm is that it loads two blocks at a time and keeps a pointer on each entry. While scanned through the entries and merging them, if one block finishes first, the remaining other

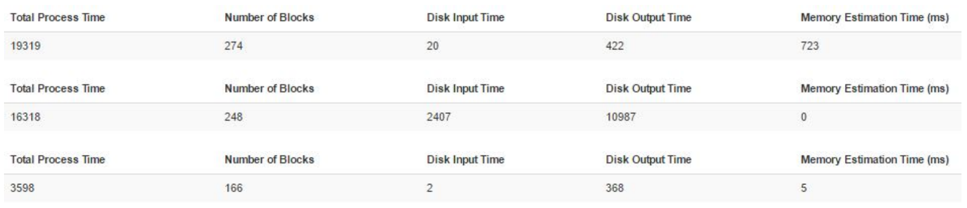
entries of the other block are kept as residual so as to be merged with an upcoming new block. Once these blocks are merged they'll be written out to disk as super blocks. In this way, the algorithm recursively builds super blocks until the entire blocks are completely merged.

2.3 Posting List Compression

Given the posting lists grow in size to accommodate the increasing document Ids, the Posting list is to be stored in compressed form for less disk space usage and increased performance with faster data transfer from disk to memory. The compression technique employed uses Variable-byte differential encoding algorithm by exploiting an external library, i.e. javaFastPFOR. This is because a higher compression ratio can be achieved by exploiting the fact that document IDs are sorted in ascending order in the postings list. Instead of compressing raw document IDs, the difference/gap between actual and previous document ID is compressed. Moreover, since gaps are small for frequent terms and large for infrequent terms, Variable byte encoding of the gap integers provide a more efficient memory allocation. In addition, important to note is that the same memory estimation technique is applied during compressed index construction to estimate the size of a dictionary block in memory and on disk before flushing to disk. Thereby, the compression resulted in less number of blocks for the final inverted index. Compressing the posting lists is demonstrated to be beneficial in the selected data collection since the document Ids are set to be large numbers to simulate real world case and that quite often most terms in the collection are very frequent with small gap size. Hence, encoding the gaps in variable bytes as opposed to integers prevents considerable memory wastage as further discussed in the evaluation section.

3 Boolean Search

This section discusses locating information from the inverted index build. The searching technique is a binary search which consists of terms as basic queries which are connected by Boolean operators (AND, OR, NOT) to return the desired set of documents. The result set of document either satisfies the query or it does not, hence this method does not allow for deciding which document is more relevant to the user among the set of returned documents.
In order to optimize the search operation, the algorithm only requires a single disk input of every block until the terms in the query are found. In other words, the algorithm terminates if, for instance, all the query terms are found in the first disk input. Moreover, before searching for the query terms, it loads a certain number of blocks until the memory is full and starts to scan the dictionary to look for the terms in the query. Once the entries



Total Process Time	Number of Blocks	Disk Input Time	Disk Output Time	Memory Estimation Time (ms)
19319	274	20	422	723

Total Process Time	Number of Blocks	Disk Input Time	Disk Output Time	Memory Estimation Time (ms)
16318	248	2407	10987	0

Total Process Time	Number of Blocks	Disk Input Time	Disk Output Time	Memory Estimation Time (ms)
3598	166	2	368	5

Figure 1: Index Construction Computational Data

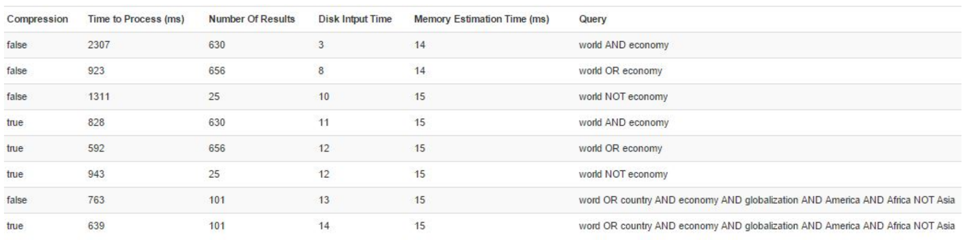
in the index for all the query terms are located with their corresponding posting lists, it initiates the boolean search operation. The boolean search supports any number of terms with the above mentioned boolean operators placed in between.
In order to ”short-circuit” the loop while finding common elements in the AND boolean operation, the algorithm first gets the posting list with the least size and iterates through it to find common document ids found in the other posting list of the second term. The second posting list, which is the inner loop is only scanned once as a pointer is placed on each posing list arrays. Likewise, the NOT operator also keeps a pointer on both posting list arrays of size n and m. Hence, while the iteration is in progress, it avoids scanning the inner loop n number of times, where n is the size of the outer loop, eliminating (n*m) complexity.

4 Evaluation Results

This section discusses the results obtained while conducting an evaluation of the system in three ways as follows.

4.1 Index construction time

This test examines the computational time taken for building the initial inverted index blocks, the merge procedure, and the elapsed time during compression as shown in figure 1. Nonetheless, it's good to note that the results obtained can be hardware architecture dependant, i.e. they vary from machine to machine. We can observe that the first step for building the initial indexes takes longer and that the compression time takes less. This is because the number of the blocks gets smaller during the merge procedure and the disk I/O time is less while compressing the final merged indexes. Hence, the overall time results to 39235 ms, i.e. roughly 39 seconds. Also to note is the overall time computed does not take into account the computational time elapsed during memory estimation conducted for computing full memory and disk block sizes.



Compression	Time to Process (ms)	Number Of Results	Disk Intput Time	Memory Estimation Time (ms)	Query
false	2307	630	3	14	world AND economy
false	923	656	8	14	world OR economy
false	1311	25	10	15	world NOT economy
true	828	630	11	15	world AND economy
true	592	656	12	15	world OR economy
true	943	25	12	15	world NOT economy
false	763	101	13	15	word OR country AND economy AND globalization AND America AND Africa NOT Asia
true	639	101	14	15	word OR country AND economy AND globalization AND America AND Africa NOT Asia

Figure 2: Boolean search results for two terms

4.2 Size of inverted index and I/O time

The results obtained for the size of the inverted index includes the initial size of blocks before merging, after merging and post compression, as in figure 1. The blocks have a size of 65 KB each for the initial 274 index blocks, 248 blocks after merging, and 166 blocks post compression. We can observe that the size of the index blocks after compression is much less and compressing the posting lists resulted to have a considerable effect in the size of final indexes on disk. Hence, the final SPIMI index has a size of 17810 KB and 10790 KB before and after compression respectively.
Moreover, the disk I/O time as shown in figure 1 shows input time to be much less than output time in general. In addition, we can also observe the effect of compression during disk I/O time since the number of total blocks is less.

4.3 Boolean query latency

The retrieval system build also measures the query latency time with the application of compression and without of the indexes. In order to make valuable comparisons between them, we decided to test the Boolean retrieval with the same queries for a fixed number of times considering only two terms in the query. The operators AND, OR and NOT have been tested with the queries ”world AND economy”, ”world OR economy” and ”world NOT economy” respectively and we have gotten the following measurements as in figure **??**. We can observe that when retrieving from the compression index lists, it results in a relatively faster search mainly due to the minimized disk I/O time given less number of blocks to search for. We have run more experiments with queries of two terms and with the different operators, but the results have showed to be similar as the experiment described above.
We have also measured the behavior of the search engine with more query terms and varying boolean operators. And the results obtained looked similar except for a small increase in the disk input time given more terms are searched for. The query tested is ”word OR country AND economy AND globalization AND America AND Africa NOT Asia”, and the corresponding

results are shown in figure **??**.

5 Conclusions

After examining all the experiments and with the experience of the construction with and without compression we can draw the following conclusions:

- Building an SPIMI index using java as language is a delicate process given that java maintains a vast amount of memory allocation for every object. Therefore, memory measure techniques and customized objects had to be implemented in order to simulate a real behavior in memory.
- The pre-processing techniques such as stop words and stemming help with the efficiency of the resulting Boolean retrieval because they optimize the search by reducing the amount of terms to look for and by getting the base or root of every term.
- The indexing of documents is a time consuming process because the pre-processing of the words, the building of the index by itself and its merging procudre can take a lot of time. Therefore, if we have a large collection of documents, the algorithm should support for multi-threading and asynchronous operations that could result in drastic efficiency during index construction, in addition to distributed indexing.
- The binary merging method has proven to be very effective since it has a logarithmic complexity which is quite better than other techniques that can have a quadratic complexity.
- It was also observed the even though our data collection is not big, compression had an effect in the size of the final index blocks since the document Ids used start from a large number and applying variable byte differential encoding had shown to have a relative impact.
- In order to optimize the search operation, *Skip lists* can also be used. Skip list, which is an extension of the posting lists, enables to lower the number of operations needed, thus processing the posting lists in a linear time.

6 External Sources

The following are external source of references to be used during every phase of this project.

6.1 Inverted index

- Manning, C. D, Raghavan, P. Schutze, H. (2009). Index Construction. In Online edition (Ed), An Introduction to Information Retrieval (pp. 67 - 84). England .

- Mahapatra, A. K. Biswas, S. (2011). Inverted indexes: Types and techniques. International Journal of Computer Science Issues, 8(4), 384 - 392.

- Zobel, J. Moffat, A. (2006). Inverted Files for Text Search Engines. ACM, 38(6), 1 - 56.

- Hadraba, A.(2015). Inverted index implementation. Czech Republic: MASARYK UNIVERSITY.

- Heinz, S. Zobel, J. (2003). Efficient Single-Pass Index Construction for Text Databases. JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE AND TECHNOLOGY, 54(8), 713 - 729.

6.2 Index Compression

- Manning, C. D, Raghavan, P. Schutze, H. (2009). Index Compression. In Online edition (Ed), An Introduction to Information Retrieval (pp. 85 - 107). England .

- Catena, M. Macdonald, C. Ounis, I. (2014). On Inverted Index Compression for Search Engine Efficiency. In Maarten de rijke et al (Eds), Advances in Information Retrieval (pp. 359-371). England: Springer International Publishing.

- Scholer et al.. (2002). Compression of Inverted Indexes For Fast Query Evaluation. ACM, S/N(S/N), 222 - 229.

6.3 Library API Documentation

- Apacheorg. (2016). Apache Lucene API. Retrieved 14 April, 2016, from https://lucene.apache.org/core/

- Lemire, D. (2014). JavaFastPFOR: A simple integer compression library in Java. Retrieved 14 April, 2016, from https://github.com/lemire/JavaFastPFOR

- Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, Ian H. Witten (2009); The WEKA Data Mining Software: An Update; SIGKDD Explorations, Volume 11, Issue 1.