dexter

Darshan Pandit & Manas Pawar

1 INTRODUCTION

Dexter is our attempt at building an Information Retreival System to enable indexing and searching over text data. While, the tool is designed keeping the New Zealand Web Corpus in mind,the tool can be used for other corpuses, with almost minimal changes and can be modified to enhance the functionality of the tool.

While the tool does not include support for storing context and offset information, some of the essential files to enable it have been included. Our failure to handle the contexts and offsets in a memory efficient way, currently leaves us with an ample scope for additions to this utility. However, that being said, the contexts and offsets can be easily handled for 4-5% of the NZ Corpus. Currently the system uses Var-Byte Compression to compress and store both the Temporary and the Final Index Structure. LRU-Caching of Inverted List Blocks has been implemented to allow for faster retrieval of the recently used terms. We are currently supporting Conjunctive Queries, implemented in a Document-at-a-time fashion. BM-25 Relevance measure is supported, with an infrastructure to support for more relevance measures.

2 PARSING

Parsing, in its truest nature, acts as the interpreter to help our system to organize the text. The parser that we use is a naive Parser, that given a page, provides tokens, that contain context information along with the token itself. The relative positions of the tokens are indicators of the offsets, which act as an essential feature in complex queries. In our case the parser generates Posting Objects.More complex parsers exist, however, designing a parser was beyond the scope of the problem, given the fact that a lot of these parsers are well suited, designed for a particular domain, or a function.

3 Indexing

3.1 Temporary Index

The design that we implement, is a two-pass indexing structure, however, the current design can be extended, to allow for a single pass structure. In the first phrase, we accept a batch of postings and create a temporary index structure. The key element in this process, as we observed, is to minimize the space occupied by the postings and th index as it would drectly imply for a larger batch sze, which as we will see further, is benfecial in other stages.

For the results demonstrated in this paper, we have processed these postings in a batch of 5Million. In a a conventional Naive approach, we end up allocating a lot unused memory to datastructures. While the problem did not surface immediately, we faced this issue when we scaled up our Corpus, to 10% of the NZ Dataset. To avoid this problem, a much standard approach of Sorting the postings and then initializing the structures, with only the required memory, assists us in the index creation process, by reducing our Garbage Collection and allocation time. While some of these optimizations may seem unnecessary, to scale the solution, it is absolute necessity to think at minutest level possible and save our space and time resources.

The process of indexing begins by sorting the collection of postings. This sorting can be either disk based or as in our implementation, an in memory approach. The issue, with dumping the postings on the disk and then sorting them, will occupy a considerable amount of Disk Bandwidth. Our approach, is to try and minimize the usage of disk, and hence we create in memory index structure from these postings list, which can be flushed onto the disk in a compressed form.

As I write this paper, we did manage to get the complete NZ DataSet indexed. However, the IndexMerge could not be completed, due to some technical issues. Following were the findings for Indexing the entire Dataset:

Total No. Of Postings	**1466505737**
Vocabulary Size	**3526382**
Total Documents	**2641034**
Documents Rejected	**151860**
Time Taken	**10327 seconds**
Merge Time is not included.	

3.2 Merging

Once the process is completed, we then move on to the task of merging these temporary index files. We implement an IndexReader structure, that can read from the indexes. The format for both the temporary as well the final index is the same, and hence we can reuse the methods implemented for the reader for this purpose. The merge-reader reads from multiple indexes present for the corpus, and can call methods on them to retrieve data from the IndexReader methods.

In the above algorithm, the heap maintains WrapperIndexEnrtries for terms provided by the readers, in a decreasing order of their documentIDs. The above process then asks flushes the docVectors

Algorithm 1 K-Way Merge

```
while $heap \neq empty$ do
    $indexEntry \leftarrow heap.poll$
    $documentVec \leftarrow idxEntry.getDocID$
    $frequencyVec \leftarrow idxEntry.getFreq$
    if $idxEntry.increamentIdx \neq -1$ then
        $heap.add \leftarrow idxEntry$
    end if
end while
```

and frequencyVectors to the disk as an index. The process is repeated for all the terms in the lexicon. The process described above can merge-K files together at a time.An optimization to current process would be to conduct the above process in parallel for n-terms, to achieve greater throughput from the system.

3.3 IndexReaders

The IndexReader structure, that we implement can read both the temporary-Indexes, as well as the final Index Structure. Such a symmetry is useful, as we can use the structure and a common cache. The structure, even allows us room for a much more scalable-distributed architecture. In such an architecture, many non-local indexes can be queried, under an abstraction, and the results be then merged to produce the final output.

The same structure can also be modified to allow room for a single-pass index, that can merge indexes and read from a temporary index for fresh documents, and then results be merged, to produce the result. Such an architecture, is highly desirable, as it allows for a better availability, a much desired feature for search engines. At this juncture, we would like to mention that much more can be done with our infrastructure, and we have created a necessary base and realized its capabilities and limitations.

3.4 Compression

Compression of the data becomes both crucial and the weapon of any such system, which deals with data of such a huge proportion. The benifits of a good compression algorithm cannot be understated, as they reduce the much dreaded, I/O bandwidth of the system. Processing power, has progressed at a much better pace than the I/O Throughputs, and hence, the cost of bearing additional processing overheads to reduce the I/O footprints is a rational decision. While, the compression in a way desrves a seperate write up, we shall briefly discuss our current compression algorithm.

Of the many other techniques to compress data, amongst the most trivial is the var-byte compression. However, the compression achieved with are significant and it stands at a good point in the complexity vs benefits graph. Other notable candidates include Goloumb Encoding and the much widely adapted P4Delta Compression. The compression ratio for them is much higher, than that of var-byte, howver, they are relatively difficult to implement, and a possible extension to this project.

A detailed algorthim for the Var-Byte Compression can be found online at this link.

The basic ideology behind achieving compression is the **Heap's Law** that instates the growth in the vocabulary growth for the corpus in proportion to its growth. The Vocabulary Size (**M**) can be expressed as:

$$M = kT^b \tag{3.1}$$

Where (**T**) is the number of terms in the Corpus.**k** & **b** are constants.

In accordance with this equation for our corpus containing 1466505737 postings, and a vocabulary size of 3526382. Assuming a b=0.5, we get the value of k=92 which is in accordance with the equation.

If this vocabulary can be represented in an efficient fashion, the reduction in the size of the corpus would be significant. This is the basic idea behind all the compression algorithms and the law holds true for most the naturally occcuring documents.

Variable_Byte Compression is much better at compressing smaller numbers, and hence, we take delta offsets, wherever possible. In our implementation, we are using **D-Gaps** for compressing DocumentIDs. It is also used to compress the offsets contained in the complex postings.

The Compressed Size of Baby Indexes for the NZ Dataset is 1.47Gb.

4 Retrieval

4.1 Relevance Measures

To compute similarity if a document with respect to a given query, relevance measures, help us differentiate the matches, based upon their function. For the given implementation, we have implement BM-25 relevance which can be expressed as

$$BM25(q,d) = \sum_{t=0}^{M} \log \frac{N - f_1 + 0.5}{f_1 + 0.5} * \frac{(k_1+1) f_{d,t}}{K + f_{d,t}} \tag{4.1a}$$

$$K = k_1 * ((1-b) + b * \frac{\|d\|}{\|d_{avg}\|}) \tag{4.1b}$$

The Relevance functions are charecterized by their complexity, their f-measures and speed. The BM-25 is a fairly complex relevance measure, requiring parameters from many structures.

Another popular measure is the Cosine Similarity Measure, which is given as:

$$Rel(q,d) = \sum_{i=0}^{m-1} \frac{w(q,t_i) * w(d,t_i)}{\sqrt{\|d\|}} \tag{4.2a}$$

$$w(q,t) = \log 1 + \frac{N}{f_t} \tag{4.2b}$$

$$w(d,t) = 1 + \log f_{d,t} \tag{4.2c}$$

The relevance measures are subchilds of scorers in our implementation and can be inititiated as an when required. We use a **Heap** to maintain the top documents for a particular query. This is typially used, to avoid resource drag on the system and is beneficial to a large extent, as the typical use case for such systems is to retrieve the top results for a query. Even when more pages are to be desired, a typical strategy is to paginate. Another relevance is the Boolean Model, in which the documents are scored merely on the number of terms present. Such a system is a very naive, but is useful in many specific scenarios, where in speed is essential and scoring is not necessarily desired. The complexity of the measure is critical, as in disjunctive queries, the candidate set can very quickly increase, and it may consume vast resources to compute the similarity. The metrics used to judge the quality of a Relevance Measure are the Precision and Recall gained. However, both of them are contrasting in nature, and a very fine balance between them is to be achieved, for a balanced system. Various ways have been devised to express the ideal ratio between them. One such metric is the F-measure. Finding a quality metric to judge a relevance measure is an active area of research. A study of the relevance functions using the metric in our case remains pending.

4.2 QUERY PROCESSING

Minimizing the candidate pool before scoring can help us avoid the expensive scoring computations. In our sytem Queries are proceesed in DAAT model.

- Document At A time *DAAT*
  - DAAT simply involves merging the sorted-postings lists and examining each document in the union only.A min-heap can be used to store the top-k documents. Whenever a new candidate document is identied it must be scored. Documents are added to the heap as they are scored and the desired documents are maintained to ensure the space requirements of the system. In this case, the cost for scoring depends upon the seek time and the scoring time. We use DAAT Strategy, in our implementation.

In addition to this, the postings can be cached using a heap structure. Our implementation involves an **LRU Cache** for indexBlocks. These indexBlocks can be cached in compressed bytes, only to be decompressed on each request, or we can cache them uncompressed, which would have a larger memory footprint, but faster access times. In our experiments on a **5% sample of NZDataset, a 3 term query 'the and but' took about 19milliseconds for disk access, and 15milliseconds for cache-based access.**

5 OBSERVATIONS & LEARNINGS

While creating a such a system at a small scale may seem to be a trivial task, but the need to scale it up brought to our notice the minute but essential considerations while implementing such a system. The excercise also made us experience the problems of accessing disk and in memory computation.

It is worth noticing the gains achieved via Compression of the indexes and the bandwidth reduction.

6 CODE DESCRIPTION

The code can be basically divided into 8 packages. Each containing some relevant Class files. We will attempt to describe the role of each package and its objects briefly.

6.1 Compression

This package is primarily responsble to handle the Compressions used in the system. Currently, it contains implementation for computing **Variable Byte Stream** for a given structure and also method to compute **Delta or d-Gaps** for a given datastructure. It also contains relevant method to decompress the files.

6.2 Indexing

The package contains the objects necessary to construct the Inverted Index. The major components of the class include the following

- PostingWriter

  This Class is responsible to write an inverted index for a collection of postings. The Class is used initially to convert the PostingList and write into a disk. A configurable parameter, is used to specify the batchsize.

- IndexReader:

  The Objects are resopnsible to read from an inverted index, and provide the necessary byteStreams to construct indexEntries. The IndexReaders implement the essential methods:

  - openList(termID)
  - nextGeq(lp,docID)
  - getFreq(lp) The methods are extensively used during merging and searching the corpus.

    The reader also supports a Cached-mode, which uses an LRU Cache to cache the IndexEntries.

- IndexMerger:

  The Class is responsible to conduct K-Way merge for the system. It basically reads a folder, for all possible Baby-Indexes and holds a reader for each of them. Then for each termID in the corpus, it gets the corresponding indexEntries, for each baby-indexes and merges them to a bigger index-entry. We take advantage of the fact that the documentID's are sorted in the indexEntries. The Class also includes a main method to initiate merging. Merging a **5% Sample of NZ Dataset** took us about **55 seconds.** As this paper is being written the merging for whole dataset has not been successfully executed by us, due to heap issue in reading the lexicon object from the file.

- Posting Postings are generated by the parser. Collectively they are processed by PostingWriter to generate indexEntries.

- BasicIndexEntry:

  These objects contain the DocumentVector, frequencyVector, and also incase of the expanded system, the contextVectors and the OffsetVectors. They are identified by their termIds. They are the individual entities of our Inverted Indexes.

6.3 Lexicon

This package contains the Lexicon Class which is actually a Map<Integer, Lexicon-Value>. Each lexiconValue, contains a list of BlockInfos, which describe the physical location of the indexEntries, and a globalFrequency counter. The Lexicon persists throughout the sessions, and is written to disks currently between different Main processes. While it works at a smaller scale, on the complete dataset, the lexicon, gets fairly large, about 550Mb for Baby-Indexes and is not loaded into the memory. The current approach as writting it to an ObjectStream to serialize needs to changed. There is a list of BlockInfos as we need to store multiple block-infos during Baby-Indexes. The Final-Lexicon however, contains only one BlockInfo currently. The approach allows room for further extension.

6.4 Parsing

This package contains the essentials to parse.

- Parser:

  The parser tokenizes the Page and generates a List<Postings>.

- NZFolderReader:

  The NZFolderReader, reads for the tar-files in a given folder and can be used to uncompress each of them into a temporary Path. It also opens NZReaders, for each volume of the Dataset. Specific to use case. Used to traverse the dataset.

- NZFileReader:

  The NZFileReader is used to read from the archives present in the uncompressed folder. It streams uncompressed data, as it reads the data files, using the index files.

6.5 queryProcessing

Contains interfaces essential for scorers, an implementation for the BM25 Scorer and Conjunctive Query.

- Conjunctive Query:

  Implements DAAT retrieval. We use a cache to maintain the top-scored-N documents. Uses the DAAT algorithm, to identify the candidateDocuments and then scores them using the Scorer specified.

  Contains a main method to initiate the Query-console-interface.

6.6 Structures

This package contains various registrars, which are either Maps or write directly to the disk, but solely serve as Classes providing mapping to their corresponding fields.

6.7 Writers

The Writers object is currently a not used package, however, contains necessary structures to enable writing of the Position and Context Vectors in a different indexFile. It is currently unused.

7 Features

- **Var Byte Compression**
- **D-Gaps for Position and DocIDs.**
- **K-Way Merge**
- **Compressed Baby-Indexes**
- **DAAT - Conjuctive Query Processing**
- **Caching of indexEntries.**
- **Using heap to maintain top-N Scores.**
- **Successfully indexed NZ-Dataset. Not able to merge them.**

8 How To Run IT.

There are 3 Classes containing the main method. The relevant parameters can specified and they are as follows:

- processes.Indexing.java
- indexing.IndexMerger.java
- queryprocessing.ConjuctiveQuery.java

The Indexing requires about 3Gb of Heap to be allocated to index over the complete NZDataset. Our Lexicon and some other structures are in memory, and grow linearly with the corpus. Merger and ConjuctiveQuery do not have specific requirement.

9 References

1. `http://cis.poly.edu/cs6913/`
2. `http://nlp.stanford.edu/IR-book/html/htmledition/variable-byte-codes-1.html`
3. http://ww2.cs.mu.oz.au/mg/