CS6200 Information Retrieval

Homework2: Indexing, Term Positions

Objective

Implement your own index that would replace the one from elasticsearch used in HW1, then index the document collection used in HW1. Your index should be able to handle large numbers of documents and terms without using excessive memory or disk I/O.

This assignment involves writing two programs:

1. A tokenizer and indexer
2. An updated version of your HW1 ranker which uses your inverted index

You have some flexibility in the choice of algorithms and file formats for this assignment. You are expected to to explain and justify your approach, any reasonable approach should work.

Task1: Tokenizing

The first step of indexing is tokenizing documents from the collection. That is, given a raw document you need to produce a sequence of *tokens*. For the purposes of this assignment, a token is a contiguous sequence of characters which matches a regular expression (of your choice) – that is, any number of letters and numbers, possibly separated by single periods in the middle. For instance, `bob` and `376` and `98.6` and `192.160.0.1` are all tokens. `123,456` and `aunt's` are not tokens (each of these examples is two tokens — why?). All alphabetic characters should be converted to lowercase during tokenization, so `bob` and `Bob` and `BOB` are all tokenized into `bob`.

You should assign a unique integer ID to each term and document in the collection. For instance, you might want to use a token's hash code as its ID. If you decide to assign IDs, you will need to be able to convert tokens into term IDs and covert doc IDs into document names in order to run queries. This will likely require you to store the maps from term to term_id and from document to doc_id in your inverted index. One way to think about the tokenization process is as a conversion from a document to a sequence of `(term_id, doc_id, position)` tuples, which needs to be stored in your inverted index.

For instance, given a document with doc_id 20:

```
The car was in the car wash.
```

the tokenizer might produce the tuples:

```
(1, 20, 1), (2, 20, 2), (3, 20, 3), (4, 20, 4), (1, 20, 5), (2, 20, 6), (5, 20, 7)
```

with the term ID map:

```
1: the
2: car
3: was
4: in
5: wash
```

Task2: Indexing

The next step is to record each document's tokens in an inverted index. The inverted index should consist of a catalog and an inverted file. The inverted list for each term should contain the following information:

- A list of IDs of the documents which contain the term, along with the TF of the term within that document and a list of positions within the document where the term occurs. (The first term in a document has position 1, the second term has position 2, etc.)

You should also store the following information.

- The total number of distinct terms (the vocabulary size) and the total number of tokens (CF) in the document collection.

- The document length of each document
- The map between document names and their IDs, if required by your design.
- The map between terms and their IDs, if required by your design.

All inverted lists/files written on the hard drive should be sorted by DocID if using doc-at-a-time technique. This will facilitate merging, in particular with mergesort.

Stemming and Stopping

Experiment with the affects of stemming and stop word removal on query performance. To do so, create two separate indexes:

- An index where tokens are not stemmed before indexing, and stopwords are removed
- An index where tokens are stemmed and stop words are removed

You should use this list of stop words, obtained from NLTK.

You may use any standard stemming library. For instance, the python `stemming` package and the Java `Weka` package contain stemmer implementations like Porter stemmer.

Performance Requirements

Your indexing algorithm should meet the following performance requirements. Your design approach should be based on the assumption that the index woud not fit in memory. You need to add a brief explanation of how you met the requirements in your report. You may also be asked during your demo to further explain that.

- If you keep partial inverted lists in memory during indexing, you have to limit the number of postings to no more than 1000. In other words, you should not process more than 1000 documents in memory at a time.
- Your final inverted index should be stored in a single (or few) file(s), no more than 20. The total size must be at most that of the size of the unindexed document collection, around 160-180MB without stopwords.
- You should be able to access the inverted list for an arbitrary term in time at most logarithmic in the vocabulary size, regardless of where that term's information is stored in the index. You should not need to find an inverted list by scanning through the entire index.
- Extra Credit Option: You are permitted to write multiple files during the indexing process, but not more than about 1,000 files total. For instance, you may not store the inverted list for each term in a separate file.

Index Compression (MS students only)

Store the index in some compressed format and decompress it as needed when accessing it. You should only need to decompress the posting lists corresponding to the query terms. For the sake of this assignment, you may use a software for compression or decompression. For instance, it may be sufficient to run inverted lists through a gzip/gunzip routine in a library.

Task3: Searching

Update your solution to HW1 to use your index instead of elasticsearch. Compare your results to those you obtained in HW1. Are they different? If so, why? You dont have to run all 5 models; one VSM, one LM, and BM25 will suffice.

Proximity Search (MS students only)

Add one retrieval model, with scoring based on proximity on query terms in the document. You can use the ideas presented in slides, or skipgrams minimum span, or other ngram matching ideas.

Some Hints

There are many ways to write an indexing algorithm. We have intentionally not specified a particular algorithm or file format.

The primary challenge is to produce a single index file which uses a variable number of bytes for each term (because their inverted lists have different lengths), without any prior knowledge about how long each list will need to be. Here are a few reasonable approaches you might consider. Your design should be scalable and therefore assume that the final index would not fit in memory.

Option 1- Required: Merging

Create partial inverted lists for all terms in a single pass through the collection. You can process the documents in batches of 1000. As each partial list is filled, append it to the end of a single large index file. When all documents have been processed, run through the inverted files a term at a time and merge the partial lists for each term. This second step can be greatly accelerated if you keep a list of the positions of all the partial lists for each term in some secondary data structure or file.

Extra Credit Option: Discontiguous Postings

Lay out your index file as a series of fixed-length records of, say, 4096 bytes each. Each record will contain a portion of the inverted list for a term. A record will consist of a header followed by a series of inverted list entries. The header will specify the term_id, the number of inverted list entries used in the record, and the file offset of the next record for the term. Records are written to the file in a single pass through the document collection, and the records for a given term are not necessarily adjacent within the index.

Extra Credit Option: Multiple passes

Make multiple passes through the document collection. In each pass, you create the inverted lists for the next 1,000 terms, each in its own file. At the end of each pass, you concatenate the new inverted lists onto the main index file (easy to concatenate the inverted files, but have to manage the catalog/offsets files)

Extra Credit

These extra problems are provided for students who wish to dig deeper into this project. Extra credit is meant to be significantly harder and more open-ended than the standard problems. We strongly recommend completing all of the above before attempting any of these problems.

Points will be awarded based on the difficulty of the solution you attempt and how far you get. You will receive no credit unless your solution is "at least half right," as determined by the graders.

EC1: Multiple Fields

Provide the ability to index multiple document fields. Index the contents of the HEAD fields for a document (if any) in addition to the TEXT fields. Update your retrieval models to query the HEAD fields as well as the TEXT fields, weighting HEAD matches higher than TEXT matches. Does this improve retrieval performance? Why?

EC2: Query Optimization

Implement and compare multiple query processing algorithms (e.g., variations of doc-at-a-time and term-at-a-time matching) to achieve the best possible query performance. Include at least one inexact query processing method. How much can you improve query speed without overly sacrificing result quality?

Rubric

Detailed rubric can be found in Canvas. Check this piazza post for HW4 that shows expected results for both PageRank and HITS

© Northeastern University, 2024, all rights reserved