Assignment 1: Indexing

Nicolai Dahl Blicher-Petersen and Daniel Jonathan Smith
{s3441163, s3361789}@student.rmit.edu.au

RMIT

1 Index Construction

When parsing the data set of hundreds of documents we search for the HEADLINE and TEXT tags, which contain content for each document. During this process the Document ID is noted when encountering the DOCID tag, and all other tags are skipped.

Document ID Assignment

Internal Document IDs are assigned sequentially as they are encountered in the data set using a separate DocIdHandler class, which maintains a dynamically sized array of "raw" Document IDs from the data set (implemented using the standard Java ArrayList collection). The internal Document ID is an integer corresponding to the array index of the "raw" Document ID within the DocIdHandler class, to allow constant time lookups.

Parsing

Parsing content character by charater is performed in accordance with the following set of rules:

1. When reaching mark-up tags such as <p>, skip them
2. When parsing a token starting with a letter, only allow letters, numbers, and hyphens as non-terminators of the token. When reaching a terminator for a token containing a hyphen, the word is analyzed and handled in the following way:
   (a) If there is only one hyphen in the word and the first part is a common prefix such as co, pre, meta, or multi, the hyphen is removed and the two parts are concatenated into one [5][4].
   (b) If there is only one hyphen in the word and the last term ends on 'ed' (e.g. case-based) the hyphen is kept [5].
   (c) Otherwise all hyphens in the token are removed and a token with n hyphens becomes $n + 1$ tokens with no hyphens.
3. When parsing a token starting with a digit, only allow letters, digits, dots, and commas as non-terminators of the token. If commas or dots are at the end of a token they are removed.
4. If the token is less than three characters long, it is thrown away.

Note that this means that the space character will most often be the terminator of a token and that we do not handle acronyms in any special way. As described by Manning et al. [3, p. 24] handling hyphens is often done in accordance with some heuristics that keep or remove hyphens based on various attributes of a token, e.g. its length. Our parser is simplified and will produce unwanted results for sequences such as "San Francisco-Los Angeles". It will, however, manage to index words like "co-operative/cooperative" under the same term ID.

Inverted Index Construction in Memory

After each document has been parsed, the tokens that were gathered are passed to the indexer module as a term list $(t, f_{d,t})$, where t denotes the term and $f_{d,t}$ denotes the term frequency, in this context the within-document frequency of the term.

The in-memory representation of the inverted index is a HashMap that maps each term t to a document frequency f_t and a list of postings of the form $(d, f_{d,t})$, where d denotes the Document ID and $f_{d,t}$ again denotes the within-document frequency of the term. This, in turn, is represented as an ArrayList, sorted by Document ID.

Since our postings list insertion method inserts new postings in sort order, it guarantees a constant-time insertion of a document where the Document ID is greater than all existing Document IDs, and a linear-time insertion in other cases. We consistently provide a constant-time insertion in our parser implementation, as Document IDs are assigned sequentially and therefore insertion into the postings list is always done at the end.

Initially we tried passing a raw term list for each document to the indexer. Aggregating the terms within the parser provides a dramatic performance improvement when adding the terms to the inverted index, however, as each term requires only a single insertion into the postings list.

Inverted Index Representation on Disk

The inverted index implementation was inspired by Manning et al. [3].

We produce a lexicon, where each entry consists of the term t, the document frequency f_t, and the byte offset in our inverted list file. These are separated by the pipe character and each term entry is separated by a newline.

The inverted list is saved on disk as a binary file consisting of non-delimited binary integer values, to be retrieved via the byte offset stored in the lexicon. If a term t occurs in document d there will be a posting of two integers $(d, f_{d,t})$ denoting the Document ID followed by the within-document frequency of t.

2 Stoplist

The stopping as well as the indexing module have been implemented using the delegation design pattern. The stopper module's sole responsibility is to check if a word passed to it is in the stoplist provided when calling the indexing program [1].

To perform constant-time lookups in the stoplist we used a standard Java HashSet of strings. We utilize the standard Java string hash function to distribute keys between the buckets, as it has been shown to have an even distribution of hash values when used with random string values [2]. Since the size and average character distribution of the stoplist is not known in advance it is necessary to use a hash function that can provide a reasonably even distribution of hash values over a random collection of strings.

When collating terms, the parser simply checks the stopper for the existence of each term in the stoplist. If the term exists in the stoplist it is discarded and the parser moves on.

3 Index Search

Parsing of Query Terms

Our index search implementation first reads in the search terms from the command line and runs them through the same parser used to construct the inverted index. This is to ensure the same tokenisation and normalisation processes are applied to the search terms. Without this step search terms provided may not match the representation of an initially identical indexed term.

Reading in the Lexicon

The lexicon of the inverted index is read into a standard Java HashMap collection, mapping a term t to term data $(f_t, p_t$, where f_t is the document frequency and p_t is the address in the inverted list. The inverted list address is the byte offset of the postings list in the inverted list file.

Implementing the lexicon as a HashMap provides a constant time lookup for each term's frequency data and inverted list address.

Reading in the Postings List

The inverted list file is opened for random access using a Java SeekableByteChannel, which allows us to seek to arbitrary points in the file and retrieve multiple bytes at a time.

Once a search term is parsed, the inverted list address and document frequency are retrieved from the lexicon hash.

To retrieve the postings list we first seek to the byte offset stored in the inverted list address for the term in the inverted list file. Each posting is made up of two integers, the internal Document ID and in-document term frequency, so two integers multiplied by the document frequency for the term are read into a buffer.

This buffer is parsed to construct a postings list consisting of postings of the form $(d, f_{d,t})$, denoting the Document ID followed by the within-document frequency of t. The postings list is passed back to the query program as a search result for further processing.

Retrieving the Raw Document ID

The raw Document ID mapping is read from the map file into the DocIdHandler class referred to in section 1. The DocIdHandler stores the raw Document IDs in an array, where the raw Document ID resides in the array index of its corresponding internal Document ID. The DocIdHandler handles this representation internally, and simply converts an internal Document ID to a raw Document ID when requested.

Once the postings list is generated, this list is stepped over. Each raw Document ID is retrieved from the DocIdHandler and printed to the command line.

4 Index Size

When indexing the sample latimes collection, our final inverted index (including the lexicon, inverted list and document ID map files) with stopwords removed is approximately 176MB, where the original collection is approximately 476MB.

This is due to several factors:

1. The original latimes collection includes markup tags and extra metadata about each document, which we discard.
2. We remove stopwords from the collection.
3. Where each term in the collection requires disk space for an entire ASCII representation of each occurrence of a term, our inverted list simply requires space for the byte representation of two integers for each document in which a term occurs. Although in the worst case, in which a term occurs only once in a single document, this in fact occupies more space on disk, in the vast majority of cases there is a significant space saving.

A Contributions

A.1 Nicolai Dahl

1. Parser
2. Stopper
3. Search term parsing
4. Processing and display of search results
5. Index/Search front end

A.2 Daniel Smith

1. Index construction
2. Index representation on disk
3. Hash function research
4. Reading lexicon and inverted list from file
5. Parsing inverted list to produce search results

A.3 Equal

1. Program architecture
2. Report

Bibliography

[1] Delegation pattern, 2013.

[2] Neil Coffey. How the string hash function works, 2012.

[3] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval.* Cambridge University Press, New York, 2008.

[4] Richard Nordquist. Common prefixes in english, 2013.

[5] Craig Trim. The art of tokenization, 2013.