# Towards a Formal Verification of B+ Trees

Nicolai Dahl Blicher-Petersen, Christian Harrington, and Morten Fangel Jensen
{ndbl, cnha, mfan}@itu.dk

IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark

## 1 Indexing

When parsing the data set of houndreds 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. 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 co, pre, meta, or multi, the hyphen is removed and the two parts are concatenated into one [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 [4].
   (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 number, only allow letters, numbers, dots, and commas as non-terminators of the token.
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/coorperative" under the same term ID.

After each document has been parsed, the tokens that were gathered are passed to the indexer module, that builds our in-memory lexicon and inverted list. The in-memory representation of the inverted index is a HashMap that maps each term to its postings list that, in turn, is represented as an ArrayList. By using these data structures we achieve a constant-time insertion for each token. We guarantee this, as insertion into the postings list is always done to the end

of the list. Note that the list will stay sorted, as we assign our own increasing Document IDs.

The inverted index implementation was inspired by Manning et al. [3]. We produce a lexicon, where each entry consists of the term, the total term frequency in the data set, and the byte offset into our inverted list where the postings for the term is kept. These are separated by the pipe character and each entry is on its own line. The inverted list is saved on disk as raw, unformatted bytes. If a term $t$ occurs in document $d$ there will be a posting $(d, f_{d,t})$ denoting the Document ID followed by the in-document frequency of $t$.

## 2 Stopping

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 that was provided [1].

To perform constant-time lookups in the stoplist we used the standard Java HashSet of strings. We utilize the standard Java string hash function for an even distribution of keys between the buckets [2].

## 3 Querying

Explain your index search implementation. As part of your explanation you need to clearly describe: • the data structures you used for your lexicon when it is held in memory • how you look up corresponding term occurrence information in your invlists file This explanation should be around a page in length, but no longer than one and a half pages.

How large is your inverted index? How does this compare with the size of the original collection? Why is it larger/smaller than the original collection?

## A Find Subtree Source

# 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] Craig Trim. The art of tokenization, 2013.