Information Retrieval and Web Search
Lecture 05

Lecture by Dr. Inbal Budowski-Tal
Typeset by Steven Karas

2018-05-02
Last edited 20:44:35 2018-05-02

Disclaimer These lecture notes are based on the lecture for the course Information Retrieval and Web Search, taught by Dr. Inbal Budowski-Tal at IDC Herzliyah in the spring semester of 2017/2018. Sections may be based on the lecture slides written by Dr. Inbal Budowski-Tal.

Agenda

- Index compression
- Term statistics
- Dictionary compression
- Postings compression

1 Recap

Last week, we talked about BSBI, which constructs an index by blocks, sorts each block, and then merges each block. We also presented a MapReduce approach, which creates postings from a document, and then assigns buckets for postings based upon the terms. Postings are collected into an inverted index by doing a sorted merge at the end.

Block based merging is done using a heap of size k, where k is the number of blocks. The heap is constructed from the first element of each block. The smallest element is popped off the heap, and the next posting from its source block is pushed in. This gives us a merge performance of $O(n \log k)$.

Dynamic indexing is an approach where the index is updated online. The simple approach of keeping the main index on disk and an auxilliary index in memory (along with a deletion log) doesn't work well in practice because the index is offline during rebuilds. Logarithmic merge keeps many indexes, each twice as large as the previous. The smallest is kept in memory, and when it overflows, it is merged with the next largest and written to disk. This is done recursively until there is no next largest.

2 Compression

Generally, compression reduces the required amount of storage, and increases the cost of transferring data. In many cases, the cost of transferring data from disk to memory (or from memory to cache) can outweigh the cost of compression. There are two general approaches to compression: lossy and lossless. Term preprocessing typically performs lossy compression, by downcasing, removing stop words, etc. We'll discuss the application of lossless compression, which preserves all information. As motivation for how various types of preprocessing compress the size of the index, a table of the reduction achieved by different steps can be found on slide 15.

2.1 Term statistics

Heap's law states that the number of unique terms is $K \cdot N^{\beta}$, where both K and β are derived empirically[1]. For English collections, K is typically between 30 and 100, where β is around 0.5. Note that Heap's law is linear in the log-log space. As an example, in the RCV1 corpus, $K = 44$, and $\beta = 0.49$.

[1] by linear regression in the log-log space

Zipf's law states that the frequency of a term is approximately the inverse of its frequency rank. In formal terms, it's an estimate on the frequency of a given term with frequency rank i: $cf_i = c \cdot i^k$, where the constant terms are derived empirically[2].

2.2 Dictionary compression

The dictionary is small compared to the postings, but we want to keep it in memory. Embedded applications or similarly constrained environments are common, and compressing it can provide better cache cohesion, which can give significantly better performance.

As an example, the uncompressed dictionary for the RCV1 corpus has 400,000 entries. If we assume each entry takes 28 bytes (20 for the term, 4 bytes for a frequency counter, and 4 bytes for a offset to the postings), then we would need 10.86 MiB. However, a fixed-length entry table isn't great, because we have a few terms that are longer than that.

If we chunk all the terms together into a text pool, and index them by offset in a table, we can reduce the bytes per entry to 11, and another 8 bytes on average for the text pool. This reduces the total memory needed to 7.25 MiB. If we chunk multiple terms into blocks and keep an offset to the text pool every k terms, and prefix the term length, we reduce the usage to 9 bytes on average for each term, and $k(4 + 4) + 3$ for each block. For our example, this reduces the memory needed to 6.77 MiB. However, this adds a constant factor to walking over the dictionary.

2.2.1 Front coding

Front coding encodes a common prefix to a block of terms in a text pool. For example:

`8automata8...10automation` ⇒ `8automat*a1◦e2◦ic3◦ion`

This reduces the memory usage for the RCV1 corpus to 5.6 MiB.

2.3 Postings compression

Non-positional postings are just a bucket of document ids. We would like to use less than the packed minimum of bits (e.g. RCV1's minimum packed is 20 bits per document id). The first approach we will consider stores strides instead of full document ids. For example, given the following postings list:

```
COMPUTER: 283154, 283159, 283202
```

we can encode this instead as the first document id and then the stride between each successive document:

```
COMPUTER: 283154, 5, 43
```

We can use variable encoding to reduce the amount of memory needed.

2.3.1 Variable Length Encoding

We need to sacrifice at least a single bit to encode the control information, so we prefer to do this in a block size of 8 bits for most implementations[3]. The most significant bit is typically set to 0 for all blocks except the last. For example:

```
     5 = 10000101
214577 = 00001101 00001100 10110001
```

2.3.2 Gamma Codes

A bitlevel code can provide even better compression. Effectively, we can represent the language of numbers as a binary tree, where left branches are 0's and right branches are 1. A special class of such trees are prefix-free, which are self-synchronizing. Gamma codes are such trees that are balanced well for smaller numbers.

To encode a number, encode the offset of a number, which is the number with the leading bit chopped off (e.g. `13 = 1101` ⇒ `101`). The length of this offset is then added as a prefix to the encoded term as a prefix:

[2]again, by linear regression in the log-log space

[3]depending on implementation and hardware, it may be better to use an alignment size of 4 bits or 32 bits.

```
13 ⇒ 1110101
```

For practical purposes, there is typically a huge performance penalty to be paid for bit manipulation and queries that aren't aligned at a word boundary, so variable length encoding is better for most purposes.

3 Next Session

References

[1] Christopher D Manning, Prabhakar Raghavan, Hinrich Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.