Introduction to Information Retrieval

CS276: Information Retrieval and Web Search
Christopher Manning and Pandu Nayak
Lecture 4: Index Compression

Last lecture – index construction

- Sort-based indexing
  - Naive in-memory inversion
  - Blocked Sort-Based Indexing (BSBI)
    - Merge sort is effective for hard disk-based sorting (avoid seeks!)
- Single-Pass In-Memory Indexing (SPIMI)
  - No global dictionary
  - Generate separate dictionary for each block
  - Don’t sort postings
    - Accumulate postings in postings lists as they occur
- Distributed indexing using MapReduce
- Dynamic indexing: Multiple indices, logarithmic merge

Today

- Collection statistics in more detail (with RCV1)
  - How big will the dictionary and postings be?
- Dictionary compression
- Postings compression

Why compression (in general)?

- Use less disk space
  - Save a little money; give users more space
- Keep more stuff in memory
  - Increases speed
- Increase speed of data transfer from disk to memory
  - [read compressed data | decompress] is faster than [read uncompressed data]
- Premise: Decompression algorithms are fast
  - True of the decompression algorithms we use

Why compression for inverted indexes?

- Dictionary
  - Make it small enough to keep in main memory
  - Make it so small that you can keep some postings lists in main memory too
- Postings file(s)
  - Reduce disk space needed
  - Decrease time needed to read postings lists from disk
  - Large search engines keep a significant part of the postings in memory.
    - Compression lets you keep more in memory
- We will devise various IR-specific compression schemes

Recall Reuters RCV1

<table>
<thead>
<tr>
<th>symbol</th>
<th>statistic</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>documents</td>
<td>800,000</td>
</tr>
<tr>
<td>L</td>
<td>avg. # tokens per doc</td>
<td>200</td>
</tr>
<tr>
<td>M</td>
<td>terms (= word types)</td>
<td>~400,000</td>
</tr>
<tr>
<td></td>
<td>avg. # bytes per token</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>(incl. spaces/punct.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>avg. # bytes per token</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>(without spaces/punct.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>avg. # bytes per term</td>
<td>7.5</td>
</tr>
<tr>
<td></td>
<td>non-positional postings</td>
<td>100,000,000</td>
</tr>
</tbody>
</table>
Index parameters vs. what we index
(details IIR Table 5.1, p.80)

<table>
<thead>
<tr>
<th>size of</th>
<th>word types (terms)</th>
<th>non-positional postings</th>
<th>positional postings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dictionary</td>
<td>non-positional index</td>
<td>positional index</td>
</tr>
<tr>
<td></td>
<td>Size (K)</td>
<td>∆%</td>
<td>cumul %</td>
</tr>
<tr>
<td>Unfiltered</td>
<td>484</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>No numbers</td>
<td>474</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>Case folding</td>
<td>392</td>
<td>-17</td>
<td>-19</td>
</tr>
<tr>
<td>30 stopwords</td>
<td>391</td>
<td>-0</td>
<td>-19</td>
</tr>
<tr>
<td>stemming</td>
<td>322</td>
<td>-17</td>
<td>-33</td>
</tr>
</tbody>
</table>

Exercise: give intuitions for all the ‘0’ entries. Why do some zero entries correspond to big deltas in other columns?

Vocabulary size vs. collection size

- How big is the term vocabulary?
  - That is, how many distinct words are there?
- Can we assume an upper bound?
  - Not really: At least $70^{20} \approx 10^{37}$ different words of length 20
- In practice, the vocabulary will keep growing with the collection size
  - Especially with Unicode 😊

Heaps’ Law

For RCV1, the dashed line $\log_{10} M = 0.49 \log_{10} T + 1.64$ is the best least squares fit. Thus, $M = 10^{0.49T+1.64} \approx 10^{1.64} = 44$ and $b = 0.49$.

Good empirical fit for Reuters RCV1!

For first 1,000,020 tokens, law predicts 38,323 terms; actually, 38,365 terms

Vocabulary size vs. collection size

- Heaps’ law: $M = kT^{b}$
- $M$ is the size of the vocabulary, $T$ is the number of tokens in the collection
- Typical values: $30 \leq k \leq 100$ and $b = 0.5$
- In a log-log plot of vocabulary size $M$ vs. $T$, Heaps’ law predicts a line with slope about $\frac{1}{2}$
  - It is the simplest possible (linear) relationship between the two in log-log space
  - $\log M = \log k + b \log T$
  - An empirical finding (“empirical law”)

Lossless vs. lossy compression

- Lossless compression: All information is preserved.
  - What we mostly do in IR.
- Lossy compression: Discard some information
  - Several of the preprocessing steps can be viewed as lossy compression: case folding, stop words, stemming, number elimination.
  - Chapter 7: Prune postings entries that are unlikely to turn up in the top $k$ list for any query.
  - Almost no loss of quality in top $k$ list.

Exercises

- What is the effect of including spelling errors, vs. automatically correcting spelling errors on Heaps’ law?
- Compute the vocabulary size $M$ for this scenario:
  - Looking at a collection of web pages, you find that there are 3000 different terms in the first 10,000 tokens and 30,000 different terms in the first 1,000,000 tokens.
  - Assume a search engine indexes a total of 20,000,000,000 ($2 \times 10^{10}$) pages, containing 200 tokens on average
  - What is the size of the vocabulary of the indexed collection as predicted by Heaps’ law?
Zipf’s law

- Heaps’ law gives the vocabulary size in collections.
- We also study the relative frequencies of terms.
- In natural language, there are a few very frequent terms and very many very rare terms.
- Zipf’s law: The $i^{th}$ most frequent term has frequency proportional to $1/i$.
- $cf_i \propto 1/i = K/i$ where $K$ is a normalizing constant
- $cf_i$ is collection frequency: the number of occurrences of the term $t_i$ in the collection.

Zipf consequences

- If the most frequent term (the) occurs $cf_1$ times
- then the second most frequent term (of) occurs $cf_2/2$ times
- the third most frequent term (and) occurs $cf_3/3$ times ...
- Equivalent: $cf_i = K/i$ where $K$ is a normalizing factor, so
  - $\log cf_i = \log K - \log i$
  - Linear relationship between $\log cf_i$ and $\log i$
- Another power law relationship

Zipf’s law for Reuters RCV1

Compression

- Now, we will consider compressing the space for the dictionary and postings. We’ll do:
  - Basic Boolean index only
  - No study of positional indexes, etc.
  - But these ideas can be extended
- We will consider compression schemes

Why compress the dictionary?

- Search begins with the dictionary
- We want to keep it in memory
- Memory footprint competition with other applications
- Embedded/mobile devices may have very little memory
- Even if the dictionary isn’t in memory, we want it to be small for a fast search startup time
- So, compressing the dictionary is important
Dictionary storage – naïve version

- Array of fixed-width entries
  - ~400,000 terms; 28 bytes/term = 11.2 MB.

<table>
<thead>
<tr>
<th>Terms</th>
<th>Freq</th>
<th>Postings ptr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>656,265</td>
<td></td>
</tr>
<tr>
<td>aachen</td>
<td>65</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td></td>
</tr>
<tr>
<td>zulu</td>
<td>221</td>
<td></td>
</tr>
</tbody>
</table>

Terms Freq Postings ptr.

20 bytes 4 bytes each

Dictionary search structure

Fixed-width terms are wasteful

- Most of the bytes in the Term column are wasted – we allot 20 bytes for 1 letter terms.
  - And we still can’t handle supercalifragilisticexpialidocious or hydrochlorofluorocarbons.
- Written English averages ~4.5 characters/word.
  - Exercise: Why is/isn’t this the number to use for estimating the dictionary size?
- Ave. dictionary word in English: ~8 characters
  - How do we use ~8 characters per dictionary term?
- Short words dominate token counts but not type average.

Compressing the term list: Dictionary-as-a-String

- Store dictionary as a (long) string of characters:
  - Pointer to next word shows end of current word
  - Hope to save up to 60% of dictionary space

<table>
<thead>
<tr>
<th>Freq</th>
<th>Postings ptr.</th>
<th>Term ptr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td></td>
<td></td>
</tr>
<tr>
<td>126</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total string length = 400K x 8B = 3.2MB

Pointers resolve 3.2M positions: log₂3.2M = 22 bits = 3 bytes

Space for dictionary as a string

- 4 bytes per term for Freq.
- 4 bytes per term for pointer to Postings.
- 3 bytes per term pointer
- Avg. 8 bytes per term in term string
- 400K terms x 19 ⇒ 7.6 MB (against 11.2MB for fixed width)

Blocking

- Store pointers to every kth term string.
  - Example below: k=4.
  - Need to store term lengths (1 extra byte)

<table>
<thead>
<tr>
<th>Freq</th>
<th>Postings ptr.</th>
<th>Term ptr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td></td>
<td></td>
</tr>
<tr>
<td>126</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Save 9 bytes on 3 pointers.

Blocking Net Gains

- Example for block size k = 4
  - Where we used 3 bytes/pointer without blocking
    - 3 x 4 = 12 bytes, now we use 3 + 4 = 7 bytes.
  - Shaved another ~0.5MB. This reduces the size of the dictionary from 7.6 MB to 7.1 MB.
  - We can save more with larger k.

Question: Why not go with larger k?
Dictionary search without blocking

- Assuming each dictionary term equally likely in query (not really so in practice!), average number of comparisons = \((1+2 \cdot 2+4 \cdot 3+4)/8 \approx 2.6\)

Exercise: what if the frequencies of query terms were non-uniform but known, how would you structure the dictionary search tree?

Dictionary search with blocking

- Binary search down to 4-term block;
  - Then linear search through terms in block.
- Blocks of 4 (binary tree), avg. = \((1+2 \cdot 2+2 \cdot 3+2 \cdot 4+5)/8 = 3\) compares

Exercises

- Estimate the space usage (and savings compared to 7.6 MB) with blocking, for block sizes of \(k = 4, 8\) and 16.

- Estimate the impact on search performance (and slowdown compared to \(k=1\)) with blocking, for block sizes of \(k = 4, 8\) and 16.

Front coding

- Front-coding:
  - Sorted words commonly have long common prefix – store differences only
  - (for last \(k-1\) in a block of \(k\))

  \[\text{automat} \rightarrow \text{automat} \text{e} \text{ic} \text{tion}\]

  Begins to resemble general string compression.

RCV1 dictionary compression summary

<table>
<thead>
<tr>
<th>Technique</th>
<th>Size in MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed width</td>
<td>11.2</td>
</tr>
<tr>
<td>Dictionary-as-String with pointers to every term</td>
<td>7.6</td>
</tr>
<tr>
<td>+ blocking, (k = 4)</td>
<td>7.1</td>
</tr>
<tr>
<td>+ blocking + front coding</td>
<td>5.9</td>
</tr>
</tbody>
</table>

POSTINGS COMPRESSION
Postings compression

- The postings file is much larger than the dictionary, factor of at least 10, often over 100 times larger
- Key desideratum: store each posting compactly.
- A posting for our purposes is a docID.
- For Reuters (800,000 documents), we would use 32 bits per docID when using 4-byte integers.
- Alternatively, we can use $\log_2 800,000 \approx 20$ bits per docID.
- Our goal: use far fewer than 20 bits per docID.

Postings: two conflicting forces

- A term like arachnocentric occurs in maybe one doc out of a million – we would like to store this posting using $\log_2 1,000,000 \approx 20$ bits.
- A term like the occurs in virtually every doc, so 20 bits/posting = 2MB is too expensive.
- Prefer 0/1 bitmap vector in this case (~100K)

Gap encoding of postings file entries

- We store the list of docs containing a term in increasing order of docID.
  - computer: 33, 47, 154, 155, 202 ...
- Consequence: it suffices to store gaps.
  - 33, 14, 107, 5, 43 ...
- Hope: most gaps can be encoded/stored with far fewer than 20 bits.
  - Especially for common words

Three postings entries

<table>
<thead>
<tr>
<th>term</th>
<th>encoding</th>
<th>postings list</th>
</tr>
</thead>
<tbody>
<tr>
<td>the</td>
<td></td>
<td>203904 1 203901 1 203900 1 ...</td>
</tr>
<tr>
<td>computer</td>
<td></td>
<td>203904 107 203904 43 ...</td>
</tr>
<tr>
<td>arachnocentric</td>
<td>250100</td>
<td>500300 204000</td>
</tr>
</tbody>
</table>

Variable length encoding

- Aim:
  - For arachnocentric, we will use ~20 bits/gap entry.
  - For the, we will use ~1 bit/gap entry.
- If the average gap for a term is $G$, we want to use $\log_2 G$ bits/gap entry.
- Key challenge: encode every integer (gap) with about as few bits as needed for that integer.
- This requires a variable length encoding
- Variable length codes achieve this by using short codes for small numbers
Introduction to Information Retrieval

Gamma codes

- We can compress better with bit-level codes
  - The Gamma code is the best known of these.
- Represent a gap $G$ as a pair length and offset
  - For example $13 \rightarrow 1101 \rightarrow 101$
- length is the length of offset
  - For $13$ (offset $101$), this is $3$.
- We encode length with unary code: $1110$.
- Gamma code of $13$ is the concatenation of length and offset: $1110101$

Gamma code examples

<table>
<thead>
<tr>
<th>number</th>
<th>length</th>
<th>offset</th>
<th>y-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>none</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>0</td>
<td>10,0</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>1</td>
<td>10,1</td>
</tr>
<tr>
<td>4</td>
<td>110</td>
<td>00</td>
<td>110,00</td>
</tr>
<tr>
<td>9</td>
<td>1110</td>
<td>001</td>
<td>1110,001</td>
</tr>
<tr>
<td>13</td>
<td>1110</td>
<td>101</td>
<td>1110,101</td>
</tr>
<tr>
<td>24</td>
<td>11110</td>
<td>1000</td>
<td>11110,1000</td>
</tr>
<tr>
<td>511</td>
<td>111111110</td>
<td>11111111</td>
<td>1111111111,111111111</td>
</tr>
<tr>
<td>1025</td>
<td>1111111110</td>
<td>000000001</td>
<td>11111111110,000000001</td>
</tr>
</tbody>
</table>

Reminder: bitwise operations

- Python (and most everything else):
  - & bitwise and; | bitwise or; ^ bitwise xor; ~ ones complement
  - << left shift bits, >> right shift; LACKS >>> zero fill right shift
- Recipes:
  - Extract 7 bits: $a \& 0x7f00 >> 8$ ; if take high-order bit add: & 0x7f
  - Combine 3 5-bit numbers: $a | b << 5 | c << 10$
  - Lookup tables rather than decoding can be faster, yet still small

Gamma code properties

- $G$ is encoded using $2 \lceil \log G \rceil + 1$ bits
  - Length of offset is $\lfloor \log G \rfloor$ bits
  - Length of length is $\lceil \log G \rceil + 1$ bits
- All gamma codes have an odd number of bits
- Almost within a factor of 2 of best possible, $\log_2 G$
- Gamma code is uniquely prefix-decodable, like VB
- Gamma code can be used for any distribution
  - Optimal for $P(n) = 1/(2^n)$
- Gamma code is parameter-free

Gamma seldom used in practice

- Machines have word boundaries – 8, 16, 32, 64 bits
  - Operations that cross word boundaries are slower
  - Compressing and manipulating at the granularity of bits can be too slow
- All modern practice is to use byte or word aligned codes
  - Variable byte encoding is a faster, conceptually simpler compression scheme, with decent compression

Variable Byte (VB) codes

- For a gap value $G$, we want to use close to the fewest bytes needed to hold $\log_2 G$ bits
- Begin with one byte to store $G$ and dedicate 1 bit in it to be a continuation bit $c$
  - If $G \leq 127$, binary-encode it in the 7 available bits and set $c = 1$
  - Else encode $G$’s lower-order 7 bits and then use additional bytes to encode the higher order bits using the same algorithm
  - At the end set the continuation bit of the last byte to $1$ (c =1) – and for the other bytes $c = 0$. 
Example

<table>
<thead>
<tr>
<th>docIDs</th>
<th>824</th>
<th>829</th>
<th>215406</th>
</tr>
</thead>
<tbody>
<tr>
<td>gaps</td>
<td>5</td>
<td></td>
<td>214577</td>
</tr>
<tr>
<td>VB code</td>
<td>00000110</td>
<td>1000101</td>
<td>00001101</td>
</tr>
</tbody>
</table>

Postings stored as the byte concatenation: 00001101011100010000101000011010000110010110001

Key property: VB-encoded postings are uniquely prefix-decodable.

For a small gap (5), VB uses a whole byte.

Other variable unit codes

- Variable byte codes are used by many real systems
  - Good low-tech blend of variable-length coding and sensitivity to computer memory alignment matches
  - Byte alignment wastes space if you have many small gaps – as gap encoding often makes
  - More modern work mainly uses the ideas:
    - Be word aligned (32 or 64 bits; even faster)
    - Encode several gaps at the same time
    - Often assume a maximum gap size, perhaps with an escape

RCV1 compression

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Size in MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>dictionary, fixed-width</td>
<td>11.2</td>
</tr>
<tr>
<td>dictionary, term pointers into string</td>
<td>7.6</td>
</tr>
<tr>
<td>with blocking, k = 4</td>
<td>7.1</td>
</tr>
<tr>
<td>with blocking &amp; front coding</td>
<td>5.9</td>
</tr>
<tr>
<td>collection (text, xml markup etc)</td>
<td>3,600.0</td>
</tr>
<tr>
<td>collection (text)</td>
<td>960.0</td>
</tr>
<tr>
<td>Term-doc incidence matrix</td>
<td>40,000.0</td>
</tr>
<tr>
<td>postings, uncompressed (32-bit words)</td>
<td>400.0</td>
</tr>
<tr>
<td>postings, uncompressed (20 bits)</td>
<td>250.0</td>
</tr>
<tr>
<td>postings, variable byte encoded</td>
<td>116.0</td>
</tr>
<tr>
<td>postings, γ-encoded</td>
<td>101.0</td>
</tr>
</tbody>
</table>

Simple-9 [Anh & Moffat, 2004]

A word-aligned, multiple number encoding scheme
How can we store several numbers in 32 bits with a format selector?

Group Variable Integer code

- Used by Google around turn of millennium....
  - Jeff Dean, keynote at WSDM 2009 and presentations at IS276
  - Encodes 4 integers in blocks of size 5–17 bytes
  - First byte: four 2-bit binary length fields
    - \( L_1 L_2 L_3 L_4 \), \( L_i \in \{1,2,3,4\} \)
    - Then, \( L_1+L_2+L_3+L_4 \) bytes (between 4–16) hold 4 numbers
    - Each number can use 8/16/24/32 bits. Max gap length ~4 billion
  - It was suggested that this was about twice as fast as VB encoding
    - Decoding gaps is much simpler – no bit masking
    - First byte can be decoded with lookup table or switch

Simple9 Encoding Scheme [Anh & Moffat, 2004]

- Encoding block: 4 bytes (32 bits)
- Most significant nibble (4 bits) describe the layout of the 28 other bits as follows:
  - 0: a single 28-bit number
  - 1: two 14-bit numbers
  - 2: three 9-bit numbers (and one spare bit)
  - 3: four 7-bit numbers
  - 4: five 5-bit numbers (and three spare bits)
  - 5: seven 4-bit numbers
  - 6: nine 3-bit numbers (and one spare bit)
  - 7: fourteen two-bit numbers
  - 8: twenty-eight one-bit numbers
- Simple16 is a variant with 5 additional (uneven) configurations
- Efficiently decoded with hand-coded decoder, using bit masks
- Extended Simple Family – idea applies to 64-bit words, etc.
Index compression summary

- We can now create an index for highly efficient Boolean retrieval that is very space efficient
- Only 4% of the total size of the collection
- Only 10-15% of the total size of the text in the collection
- We've ignored positional information
- Hence, space savings are less for indexes used in practice
  - But techniques substantially the same

Resources for today’s lecture

- IIR 5
- MG 3.3, 3.4.
  - Variable byte codes
  - Word aligned codes