Introduction to Information Retrieval

CS276: Information Retrieval and Web Search
Christopher Manning and Prabhakar Raghavan

Lecture 5: Index Compression
Last lecture – index construction

- Sort-based indexing
  - Naïve in-memory inversion
  - Blocked Sort-Based Indexing
    - Merge sort is effective for disk-based sorting (avoid seeks!)

- Single-Pass In-Memory Indexing
  - 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Index</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>11</th>
<th>31</th>
<th>45</th>
<th>173</th>
<th>174</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brutus</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>11</td>
<td>31</td>
<td>45</td>
<td>173</td>
<td>174</td>
</tr>
<tr>
<td>Caesar</td>
<td></td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>16</td>
<td>57</td>
<td>132</td>
</tr>
<tr>
<td>Calpurnia</td>
<td></td>
<td>2</td>
<td>31</td>
<td>54</td>
<td>101</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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
  - Saves a little money
- Keep more stuff in memory
  - Increases speed
- Increase speed of data transfer from disk to memory
  - \([\text{read compressed data | decompress}]\) is faster than \([\text{read uncompressed data}]\)
- 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>Size (K)</td>
<td>Size (K)</td>
</tr>
<tr>
<td></td>
<td>Δ%</td>
<td>Δ%</td>
<td>Δ%</td>
</tr>
<tr>
<td></td>
<td>cumul %</td>
<td>cumul %</td>
<td>cumul %</td>
</tr>
<tr>
<td>Unfiltered</td>
<td>484</td>
<td>109,971</td>
<td>197,879</td>
</tr>
<tr>
<td>No numbers</td>
<td>474</td>
<td>100,680</td>
<td>179,158</td>
</tr>
<tr>
<td>Case folding</td>
<td>392</td>
<td>96,969</td>
<td>179,158</td>
</tr>
<tr>
<td>30 stopwords</td>
<td>391</td>
<td>83,390</td>
<td>121,858</td>
</tr>
<tr>
<td>150 stopwords</td>
<td>391</td>
<td>67,002</td>
<td>94,517</td>
</tr>
<tr>
<td>stemming</td>
<td>322</td>
<td>63,812</td>
<td>94,517</td>
</tr>
</tbody>
</table>

Exercise: give intuitions for all the ‘0’ entries. Why do some zero entries correspond to big deltas in other columns?
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.
- Chap/Lecture 7: Prune postings entries that are unlikely to turn up in the top $k$ list for any query.
  - Almost no loss quality for top $k$ list.
Vocabulary 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 $70^{20} = 10^{37}$ different words of length 20
- In practice, the vocabulary will keep growing with the collection size
  - Especially with Unicode 😊
Vocabulary 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 $b \approx 0.5$
- 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 relationship between the two in log-log space
  - An empirical finding ("empirical law")
Heaps’ Law

For RCV1, the dashed line

\[ \log_{10} M = 0.49 \log_{10} T + 1.64 \]

is the best least squares fit. Thus, \( M = 10^{1.64} T^{0.49} \) so \( k = 10^{1.64} \approx 44 \) 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.
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 $c_{f_1}$ times
  - then the second most frequent term (of) occurs $c_{f_1}/2$ times
  - the third most frequent term (and) occurs $c_{f_1}/3$ times ...
- Equivalent: $c_{f_i} = K/i$ where $K$ is a normalizing factor, so
  - $\log c_{f_i} = \log K - \log i$
  - Linear relationship between $\log c_{f_i}$ 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
  - Basic Boolean index only
  - No study of positional indexes, etc.
  - We will consider compression schemes
DICTIONARY COMPRESSION
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 - first cut

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

### Dictionary search structure

<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>

20 bytes 4 bytes each
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: \( \log_2 3.2M = 22 \) 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 $\Rightarrow$ 7.6 MB (against 11.2MB for fixed width)

Now avg. 11 bytes/term, not 20.
Blocking

- Store pointers to every $k$th 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>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

...7ystile9syzygetic8syzygial6syzygy11szaibelyite8szczecin9szomo....

Save 9 bytes on 3 pointers.

Lose 4 bytes on term lengths.
Net

- Example for block size $k = 4$
- Where we used 3 bytes/pointer without blocking
  - $3 \times 4 = 12$ bytes,

now we use $3 + 4 = 7$ bytes.

Shaved another $\sim 0.5$ MB. This reduces the size of the dictionary from 7.6 MB to 7.1 MB. We can save more with larger $k$.

*Why not go with larger $k$?*
Exercise

- Estimate the space usage (and savings compared to 7.6 MB) with blocking, for block sizes of $k = 4, 8$ and $16$. 
Dictionary search without blocking

- Assuming each dictionary term equally likely in query (not really so in practice!), average number of comparisons
  \[ \frac{1+2\cdot2+4\cdot3+4}{8} \approx 2.6 \]

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. = \( \frac{1+2\cdot2+2\cdot3+2\cdot4+5}{8} = 3 \) compares
Exercise

- 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 $k$)
  
  $8\text{automata}8\text{automate}9\text{automatic}10\text{automation}$

  $\rightarrow 8\text{automat}^*a1\diamond e2\diamond ic3\diamond ion$

  Encodes $\text{automat}$

  Extra length beyond $\text{automat}$.  

Begin 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>Also, blocking $k = 4$</td>
<td>7.1</td>
</tr>
<tr>
<td>Also, 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.
- 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 a lot less 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 $\log_2 1M \sim 20$ bits.

- A term like *the* occurs in virtually every doc, so 20 bits/posting is too expensive.
  - Prefer 0/1 bitmap vector in this case
Postings file entry

- We store the list of docs containing a term in increasing order of docID.
  - **computer**: 33, 47, 154, 159, 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.
### Three postings entries

<table>
<thead>
<tr>
<th></th>
<th>encoding</th>
<th>postings list</th>
</tr>
</thead>
<tbody>
<tr>
<td>THE</td>
<td>docIDs</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>gaps</td>
<td>283042, 283043, 283044, 283045, ...</td>
</tr>
<tr>
<td>COMPUTER</td>
<td>docIDs</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>gaps</td>
<td>283047, 283154, 283159, 283202, ...</td>
</tr>
<tr>
<td>ARACHNOCENTRIC</td>
<td>docIDs</td>
<td>252000, 500100</td>
</tr>
<tr>
<td></td>
<td>gaps</td>
<td>252000, 248100</td>
</tr>
</tbody>
</table>
Variable length encoding

- **Aim:**
  - For *arachnocentric*, we will use \(\sim 20 \) bits/gap entry.
  - For *the*, we will use \(\sim 1 \) bit/gap entry.
- If the average gap for a term is \(G\), we want to use \(\sim \log_2 G\) 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
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>docID</th>
<th>824</th>
<th>829</th>
<th>215406</th>
</tr>
</thead>
<tbody>
<tr>
<td>gaps</td>
<td></td>
<td>5</td>
<td>214577</td>
</tr>
<tr>
<td>VB code</td>
<td>00000110 10111000</td>
<td>10000101 00001100 10110001</td>
<td></td>
</tr>
</tbody>
</table>

**Postings stored as the byte concatenation**

000001101011100010000101000011010000110010110001

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

**For a small gap (5), VB uses a whole byte.**
Other variable unit codes

- Instead of bytes, we can also use a different “unit of alignment”: 32 bits (words), 16 bits, 4 bits (nibbles).
- Variable byte alignment wastes space if you have many small gaps – nibbles do better in such cases.
- Variable byte codes:
  - Used by many commercial/research systems
  - Good low-tech blend of variable-length coding and sensitivity to computer memory alignment matches (vs. bit-level codes, which we look at next).
- There is also recent work on word-aligned codes that pack a variable number of gaps into one word
Unary code

- Represent \( n \) as \( n \) 1s with a final 0.
- Unary code for 3 is 1110.
- Unary code for 40 is
  \[
  1111111111111111111111111111111111111111110
  \]
- Unary code for 80 is:
  \[
  111111111111111111111111111111111111111111111
  111111111111111111111111111111111111111111110
  \]
- This doesn’t look promising, but....
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*
- *offset* is $G$ in binary, with the leading bit cut off
  - 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>γ-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>none</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>1111111</td>
<td>111111110,11111111</td>
</tr>
<tr>
<td>1025</td>
<td>11111111110</td>
<td>000000001</td>
<td>11111111110,0000000001</td>
</tr>
</tbody>
</table>
Gamma code properties

- $G$ is encoded using $2 \lfloor \log G \rfloor + 1$ bits
  - Length of offset is $\lfloor \log G \rfloor$ bits
  - Length of length is $\lfloor \log G \rfloor + 1$ 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
- 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 slow
- Variable byte encoding is aligned and thus potentially more efficient
- Regardless of efficiency, variable byte is conceptually simpler at little additional space cost
# 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>
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
- However, 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