Introduction to Information Retrieval

Index Compression

Ruixuan Li
Intelligent and Distributed Computing Laboratory
Huazhong University of Science and Technology
http://idc.hust.edu.cn/~rxli/
October, 2012
**Data compression**

- Lossless versus lossy compression
  - Lossy
  - Lossless
- Theory
  - Machine learning
  - Data differencing
- Uses
  - Audio
  - Video

**Why compression in IR?**

- 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

**Current hardware in Google**

- Specifications:
  - In 2002, upwards of 15,000 servers ranging from 533 MHz Intel Celeron to dual 1.4 GHz Intel Pentium III.
  - One or more 80 GB hard disks per server (2003)
  - 2–4 GB of memory per machine (2004)
  - A 2005 estimate by Paul Strassmann has 200,000 servers, while unspecified sources claimed this number to be upwards of 450,000 in 2006.
  - ~16 GB RAM, 2 TB disk space per machine (2009)

**Reuters RCV1 statistics**

<table>
<thead>
<tr>
<th>statistic</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>documents</td>
<td>800,000</td>
</tr>
<tr>
<td>avg. # tokens per doc</td>
<td>200</td>
</tr>
<tr>
<td>terms (= word types)</td>
<td>400,000</td>
</tr>
<tr>
<td>avg. # bytes per token</td>
<td>6</td>
</tr>
<tr>
<td>(incl. spaces/punct.)</td>
<td>6</td>
</tr>
<tr>
<td>avg. # bytes per token</td>
<td>4.5</td>
</tr>
<tr>
<td>(without spaces/punct.)</td>
<td></td>
</tr>
<tr>
<td>avg. # bytes per term</td>
<td>7.5</td>
</tr>
<tr>
<td>non-positional postings</td>
<td>100,000,000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>statistic</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>word types (terms)</td>
<td>400,000</td>
</tr>
<tr>
<td>non-positional postings</td>
<td></td>
</tr>
<tr>
<td>dictionary</td>
<td></td>
</tr>
<tr>
<td>size (K)</td>
<td>484</td>
</tr>
<tr>
<td>∆%</td>
<td>-2%</td>
</tr>
<tr>
<td>cumul %</td>
<td>197,879</td>
</tr>
<tr>
<td>non-positional index</td>
<td></td>
</tr>
<tr>
<td>size (K)</td>
<td>484</td>
</tr>
<tr>
<td>∆%</td>
<td>-2%</td>
</tr>
<tr>
<td>cumul %</td>
<td>197,879</td>
</tr>
<tr>
<td>positional postings</td>
<td></td>
</tr>
<tr>
<td>size (K)</td>
<td>484</td>
</tr>
<tr>
<td>∆%</td>
<td>-2%</td>
</tr>
<tr>
<td>cumul %</td>
<td>197,879</td>
</tr>
</tbody>
</table>

\[
\text{Unfiltered: } (478-484)*100%/484 = -2\%,
\text{Case folding: } (322-484)*100%/484 = -33\%
\]

**Index parameters vs. what we index**

<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>dictionary</td>
<td>size (K)</td>
<td>∆%</td>
<td>cumul %</td>
</tr>
<tr>
<td>Unfiltered</td>
<td>484</td>
<td>-2%</td>
<td>197,879</td>
</tr>
<tr>
<td>No numbers</td>
<td>474</td>
<td>-2%</td>
<td>100,680</td>
</tr>
<tr>
<td>Case folding</td>
<td>362</td>
<td>-17%</td>
<td>96,969</td>
</tr>
<tr>
<td>30 stopwords</td>
<td>391</td>
<td>-9%</td>
<td>83,390</td>
</tr>
<tr>
<td>150 stopwords</td>
<td>391</td>
<td>-9%</td>
<td>67,002</td>
</tr>
<tr>
<td>stemming</td>
<td>322</td>
<td>-17%</td>
<td>63,812</td>
</tr>
</tbody>
</table>

(478-484)*100%/484 = -2%, (322-484)*100%/484 = -33%
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

- 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”)

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 \( 1/i \).
  - \( cf_i \approx 1/i = K/i \) where \( K \) is a normalizing constant
  - \( cf_i \) is collection frequency: the number of occurrences of the term \( t_i \) in the collection.

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

Zipf’s consequences

- If the most frequent term (the) occurs \( cf_1 \) times
  - then the second most frequent term (of) occurs \( cf_2 \) times
  - the third most frequent term (and) occurs \( cf_3 \) 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
Compressing the dictionary is important because:

- 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.

Fixed-width terms are wasteful:

- Most of the bytes in the Term column are wasted – we allot 20 bytes for 1-letter terms.
- Written English averages ~4.5 characters/word.
- Ave. dictionary word in English: ~8 characters

Dictionary storage - first cut:

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

Overview:

- Recap
- Introduction
- Dictionary compression
- Posting 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.

Overview:

- Recap
- Introduction
- Dictionary compression
- Posting compression

Dictionary storage - first cut:

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

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.
- 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

- 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.

```
systezygeticsyzygialsyzygyszaibelyiteszczecinszomo...
```

- 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.

Freq. | Postings ptr. | Term ptr. |
------|---------------|-----------|
33    |               |           |
29    |               |           |
44    |               |           |
126   |               |           |

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

Pointers resolve 3.2M positions: \( \log_{2}3.2M = 22 \text{ bits} = 3 \text{ bytes} \)

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)

Blocking

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

Freq. | Postings ptr. | Term ptr. |
------|---------------|-----------|
7     |               |           |
8     |               |           |
11    |               |           |

Save 9 bytes on 3 pointers.
Lose 4 bytes on term lengths.

Dictionary search without blocking

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

Dictionary search with blocking

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

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

$$\text{automata} \rightarrow \text{automate} \rightarrow \text{automatic} \rightarrow \text{automation}$$

Encodes $\text{automat}$.

Extra length beyond $\text{automat}$.

Begins to resemble general string compression.

RCV1 dictionary compression

<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>

Overview

- Recap
- Introduction
- Dictionary compression
- Posting 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 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 $\log_2 1,000,000 \sim 20$ 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, 14, 159, 202 ...
  - Consequence: it suffices to store gaps.
    - 33, 14, 107, 43 ...
  - Hope: most gaps can be encoded/stored with far fewer than 20 bits.
Three postings entries

<table>
<thead>
<tr>
<th>term</th>
<th>posting bit</th>
<th>254042</th>
<th>20843</th>
<th>26194</th>
<th>36204</th>
</tr>
</thead>
<tbody>
<tr>
<td>arachnocentric</td>
<td>gap</td>
<td>63</td>
<td>102</td>
<td>5</td>
<td>43</td>
</tr>
<tr>
<td>the</td>
<td>gap</td>
<td>26890</td>
<td>589100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>postings</td>
<td>gap</td>
<td>25090</td>
<td>268000</td>
<td></td>
<td></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.

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

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

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
**Gamma code properties**

- $G$ is encoded using $2 \lceil \log G \rceil + 1$ bits
  - Length of offset is $\lceil \log G \rceil$ 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
- 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>6.9</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, $\gamma$-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.

**D-gap compression**

- In certain cases, bit blocks will frequently have a non-random bit distribution pattern. Here’s an example:
  - 0001000111001111
- One of the most popular is a list of integers, each representing 1 bit. For example:
  - { 3, 7, 8, 9, 12, 13, 14, 15, 16 }
- Another common way of representing ascending sequences is by using the method of D-Gaps.
Resources for today’s lecture

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