Introduction to Information Retrieval
http://informationretrieval.org

IIR 5: Index Compression

Hinrich Schütze

Center for Information and Language Processing, University of Munich

2014-04-17
Overview

1 Recap
2 Compression
3 Term statistics
4 Dictionary compression
5 Postings compression
Outline

1. Recap
2. Compression
3. Term statistics
4. Dictionary compression
5. Postings compression
Blocked Sort-Based Indexing

![Diagram showing postings to be merged and merged postings]

- **Postings to be merged**
  - **brutus d3**
  - **caesar d4**
  - **noble d3**
  - **with d4**

- **Merged postings**
  - **brutus d2**
  - **brutus d3**
  - **caesar d1**
  - **caesar d4**
  - **julius d1**
  - **killed d2**
  - **noble d2**
  - **with d4**

Disk
Single-pass in-memory indexing

- Abbreviation: SPIMI

- Key idea 1: Generate separate dictionaries for each block – no need to maintain term-termID mapping across blocks.

- Key idea 2: Don’t sort. Accumulate postings in postings lists as they occur.

- With these two ideas we can generate a complete inverted index for each block.

- These separate indexes can then be merged into one big index.
SPIMI-Invert

SPIMI-Invert(token_stream)
1 output_file ← NEWFILE()
2 dictionary ← NEWHASH()
3 while (free memory available)
4 do token ← next(token_stream)
5 if term(token) ∉ dictionary
6 then postings_list ← ADDTODictionary(dictionary,term(token))
7 else postings_list ← GETPostingsList(dictionary,term(token))
8 if full(postings_list)
9 then postings_list ← DOUBLEPostingsList(dictionary,term(token))
10 AddToPostingsList(postings_list,docID(token))
11 sorted_terms ← SORTTerms(dictionary)
12 WRITEBlockToDisk(sorted_terms,dictionary,output_file)
13 return output_file
MapReduce for index construction

- Recap
- Compression
- Term statistics
- Dictionary compression
- Postings compression

- MapReduce for index construction

- Splits
- Assign
- Master
- Assign
- Postings

- Parser
- A-f g-p q-z
- Inverter
- A-f

- Parser
- A-f g-p q-z
- Inverter
- G-p

- Parser
- A-f g-p q-z
- Inverter
- Q-z

- Map phase
- Segment files
- Reduce phase
Dynamic indexing: Simplest approach

- Maintain **big main index on disk**
- New docs go into **small auxiliary index in memory**.
- Search across both, merge results
- Periodically, merge auxiliary index into big index
Take-away today

For each term \( t \), we store a list of all documents that contain \( t \).

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

\[ \text{dictionary} \quad \text{postings file} \]
### Take-away today

For each term $t$, we store a list of all documents that contain $t$.

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

- **Motivation for compression in information retrieval systems**
Take-away today

- Motivation for compression in information retrieval systems
- How can we compress the dictionary component of the inverted index?
Take-away today

- Motivation for compression in information retrieval systems
- How can we compress the **dictionary** component of the inverted index?
- How can we compress the **postings** component of the inverted index?
Take-away today

- Motivation for compression in information retrieval systems
- How can we compress the dictionary component of the inverted index?
- How can we compress the postings component of the inverted index?
- Term statistics: how are terms distributed in document collections?
Outline

1. Recap
2. Compression
3. Term statistics
4. Dictionary compression
5. Postings compression
Why compression? (in general)
Why compression? (in general)

- Use less disk space (saves money)
Why compression? (in general)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)
Why compression? (in general)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)
- Increase speed of transferring data from disk to memory (again, increases speed)
Why compression? (in general)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)
- Increase speed of transferring data from disk to memory (again, increases speed)
  - [read compressed data and decompress in memory] is faster than [read uncompressed data]
Why compression? (in general)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)
- Increase speed of transferring data from disk to memory (again, increases speed)
  - [read compressed data and decompress in memory]
    is faster than
  - [read uncompressed data]
- Premise: Decompression algorithms are fast.
Why compression? (in general)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)
- Increase speed of transferring data from disk to memory (again, increases speed)
  - [read compressed data and decompress in memory] is faster than [read uncompressed data]
- Premise: Decompression algorithms are fast.
- This is true of the decompression algorithms we will use.
Why compression? (in general)

- Use less disk space (saves money)
- Keep more stuff in memory (increases speed)
- Increase speed of transferring data from disk to memory (again, increases speed)
  - [read compressed data and decompress in memory] is faster than [read uncompressed data]
- Premise: Decompression algorithms are fast.
- This is true of the decompression algorithms we will use.
Why compression in information retrieval?
Why compression in information retrieval?

- First, we will consider space for dictionary
Why compression in information retrieval?

- First, we will consider space for dictionary
  - Main motivation for dictionary compression: make it small enough to keep in main memory
Why compression in information retrieval?

First, we will consider space for dictionary
- Main motivation for dictionary compression: make it small enough to keep in main memory

Then for the postings file
Why compression in information retrieval?

- First, we will consider space for dictionary
  - Main motivation for dictionary compression: make it small enough to keep in main memory
- Then for the postings file
  - Motivation: reduce disk space needed, decrease time needed to read from disk
Why compression in information retrieval?

- First, we will consider space for dictionary
  - Main motivation for dictionary compression: make it small enough to keep in main memory

- Then for the postings file
  - Motivation: reduce disk space needed, decrease time needed to read from disk
  - Note: Large search engines keep significant part of postings in memory
Why compression in information retrieval?

First, we will consider space for dictionary
- Main motivation for dictionary compression: make it small enough to keep in main memory

Then for the postings file
- Motivation: reduce disk space needed, decrease time needed to read from disk
- Note: Large search engines keep significant part of postings in memory

We will devise various compression schemes for dictionary and postings.
Lossy vs. lossless compression
Lossy vs. lossless compression

- Lossy compression: Discard some information
Lossy vs. lossless compression

- Lossy compression: Discard some information
- Several of the preprocessing steps we frequently use can be viewed as lossy compression:
Lossy vs. lossless compression

- Lossy compression: Discard some information
- Several of the preprocessing steps we frequently use can be viewed as lossy compression:
  - downcasing, stop words, porter, number elimination
Lossy vs. lossless compression

- **Lossy compression**: Discard some information
- Several of the preprocessing steps we frequently use can be viewed as lossy compression:
  - downcasing, stop words, porter, number elimination
- **Lossless compression**: All information is preserved.
Lossy vs. lossless compression

- **Lossy compression**: Discard some information
- Several of the preprocessing steps we frequently use can be viewed as lossy compression:
  - downcasing, stop words, porter, number elimination
- **Lossless compression**: All information is preserved.
  - What we mostly do in index compression
Outline

1 Recap
2 Compression
3 Term statistics
4 Dictionary compression
5 Postings compression
Model collection: The Reuters collection

<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. # word tokens per document</td>
<td>200</td>
</tr>
<tr>
<td>M</td>
<td>word types</td>
<td>400,000</td>
</tr>
<tr>
<td></td>
<td>avg. # bytes per word token (incl. spaces/punct.)</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>avg. # bytes per word token (without spaces/punct.)</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>avg. # bytes per word type</td>
<td>7.5</td>
</tr>
<tr>
<td>T</td>
<td>non-positional postings</td>
<td>100,000,000</td>
</tr>
</tbody>
</table>
Effect of preprocessing for Reuters

<table>
<thead>
<tr>
<th>size of</th>
<th>word types (terms)</th>
<th>non-positional postings</th>
<th>positional postings (word tokens)</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  Δ cml</td>
<td>size  Δ cml</td>
<td>size  Δ cml</td>
</tr>
<tr>
<td>unfiltered</td>
<td>484,494</td>
<td>109,971,179</td>
<td>197,879,290</td>
</tr>
<tr>
<td>no numbers</td>
<td>473,723 -2 -2</td>
<td>100,680,242 -8 -8</td>
<td>179,158,204 -9 -9</td>
</tr>
<tr>
<td>case folding</td>
<td>391,523 -17 -19</td>
<td>96,969,056 -3 -12</td>
<td>179,158,204 -0 -9</td>
</tr>
<tr>
<td>30 stopw’s</td>
<td>391,493 -0 -19</td>
<td>83,390,443 -14 -24</td>
<td>121,857,825 -31 -38</td>
</tr>
<tr>
<td>150 stopw’s</td>
<td>391,373 -0 -19</td>
<td>67,001,847 -30 -39</td>
<td>94,516,599 -47 -52</td>
</tr>
<tr>
<td>stemming</td>
<td>322,383 -17 -33</td>
<td>63,812,300 -4 -42</td>
<td>94,516,599 -0 -52</td>
</tr>
</tbody>
</table>

Explain differences between numbers non-positional vs positional:
-3 vs -0, -14 vs -31, -30 vs -47, -4 vs -0
How big is the term vocabulary?
How big is the term vocabulary?

- That is, how many distinct words are there?
How big is the term vocabulary?

- That is, how many distinct words are there?
- Can we assume there is an upper bound?
How big is the term vocabulary?

- That is, how many distinct words are there?
- Can we assume there is an upper bound?
- Not really: At least $70^{20} \approx 10^{37}$ different words of length 20.
How big is the term vocabulary?

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

- That is, how many distinct words are there?
- Can we assume there is an upper bound?
- Not really: At least $70^{20} \approx 10^{37}$ different words of length 20.
- The vocabulary will keep growing with collection size.
- Heaps’ law: $M = kT^b$
How big is the term vocabulary?

- That is, how many distinct words are there?
- Can we assume there is an upper bound?
- Not really: At least $70^{20} \approx 10^{37}$ different words of length 20.
- The vocabulary will keep growing with collection size.
- Heaps’ law: $M = kT^b$
- $M$ is the size of the vocabulary, $T$ is the number of tokens in the collection.
How big is the term vocabulary?

- That is, how many distinct words are there?
- Can we assume there is an upper bound?
- Not really: At least $70^{20} \approx 10^{37}$ different words of length 20.
- The vocabulary will keep growing with 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 for the parameters $k$ and $b$ are: $30 \leq k \leq 100$ and $b \approx 0.5$. 
How big is the term vocabulary?

- That is, how many distinct words are there?
- Can we assume there is an upper bound?
- Not really: At least $70^{20} \approx 10^{37}$ different words of length 20.
- The vocabulary will keep growing with 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 for the parameters $k$ and $b$ are: $30 \leq k \leq 100$ and $b \approx 0.5$.
- Heaps’ law is linear in log-log space.
How big is the term vocabulary?

- That is, how many distinct words are there?
- Can we assume there is an upper bound?
- Not really: At least $70^{20} \approx 10^{37}$ different words of length 20.
- The vocabulary will keep growing with 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 for the parameters $k$ and $b$ are: $30 \leq k \leq 100$ and $b \approx 0.5$.
- Heaps’ law is linear in log-log space.
  - It is the simplest possible relationship between collection size and vocabulary size in log-log space.
How big is the term vocabulary?

- That is, how many distinct words are there?
- Can we assume there is an upper bound?
- Not really: At least $70^{20} \approx 10^{37}$ different words of length 20.
- The vocabulary will keep growing with 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 for the parameters $k$ and $b$ are: $30 \leq k \leq 100$ and $b \approx 0.5$.
- Heaps’ law is linear in log-log space.
  - It is the simplest possible relationship between collection size and vocabulary size in log-log space.
  - Empirical law
Heaps’ law for Reuters

Vocabulary size $M$ as a function of collection size $T$ (number of tokens) for Reuters-RCV1. For these data, the dashed line $\log_{10} M = 0.49 \times \log_{10} T + 1.64$ is the best least squares fit. Thus, $M = 10^{1.64} T^{0.49}$ and $k = 10^{1.64} \approx 44$ and $b = 0.49$. 
Empirical fit for Reuters
Empirical fit for Reuters

- Good, as we just saw in the graph.
Empirical fit for Reuters

- Good, as we just saw in the graph.
- Example: for the first 1,000,020 tokens Heaps’ law predicts 38,323 terms:

\[ 44 \times 1,000,020^{0.49} \approx 38,323 \]
Empirical fit for Reuters

- Good, as we just saw in the graph.
- Example: for the first 1,000,020 tokens Heaps’ law predicts 38,323 terms:
  \[ 44 \times 1,000,020^{0.49} \approx 38,323 \]
- The actual number is 38,365 terms, very close to the prediction.
Empirical fit for Reuters

- Good, as we just saw in the graph.
- Example: for the first 1,000,020 tokens Heaps’ law predicts 38,323 terms:

\[ 44 \times 1,000,020^{0.49} \approx 38,323 \]

- The actual number is 38,365 terms, very close to the prediction.
- Empirical observation: fit is good in general.
Exercise

1. What is the effect of including spelling errors vs. automatically correcting spelling errors on Heaps’ law?

2. Compute vocabulary size $M$
   - 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?
<table>
<thead>
<tr>
<th>Recap</th>
<th>Compression</th>
<th>Term statistics</th>
<th>Dictionary compression</th>
<th>Postings compression</th>
</tr>
</thead>
</table>

Zipf’s law
Zipf’s law

- Now we have characterized the growth of the vocabulary in collections.
Zipf’s law

- Now we have characterized the growth of the vocabulary in collections.
- We also want to know how many frequent vs. infrequent terms we should expect in a collection.
Now we have characterized the growth of the vocabulary in collections.

We also want to know how many frequent vs. infrequent terms we should expect in a collection.

In natural language, there are a few very frequent terms and very many very rare terms.
Zipf’s law

- Now we have characterized the growth of the vocabulary in collections.
- We also want to know how many frequent vs. infrequent terms we should expect in a collection.
- 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 $cf_i$ proportional to $1/i$. 
Now we have characterized the growth of the vocabulary in collections.

We also want to know how many frequent vs. infrequent terms we should expect in a collection.

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 $cf_i$ proportional to $1/i$.

$cf_i \propto \frac{1}{i}$
Now we have characterized the growth of the vocabulary in collections.

We also want to know how many frequent vs. infrequent terms we should expect in a collection.

In natural language, there are a few very frequent terms and very many very rare terms.

Zipf’s law: The $i^{\text{th}}$ most frequent term has frequency $\text{cf}_i$ proportional to $1/i$.

$\text{cf}_i \propto \frac{1}{i}$

$\text{cf}_i$ is collection frequency: the number of occurrences of the term $t_i$ in the collection.
Zipf’s law
Zipf’s law

- **Zipf’s law**: The $i^{th}$ most frequent term has frequency proportional to $1/i$.
- $cf_i \propto \frac{1}{i}$
- $cf$ is collection frequency: the number of occurrences of the term in the collection.
- So if the most frequent term (*the*) occurs $cf_1$ times, then the second most frequent term (*of*) has half as many occurrences $cf_2 = \frac{1}{2}cf_1 \ldots$
Zipf’s law: The $i^{th}$ most frequent term has frequency proportional to $1/i$.

$\text{cf}_i \propto \frac{1}{i}$

$\text{cf}$ is collection frequency: the number of occurrences of the term in the collection.

So if the most frequent term (the) occurs $\text{cf}_1$ times, then the second most frequent term (of) has half as many occurrences $\text{cf}_2 = \frac{1}{2} \text{cf}_1 \ldots$

$\ldots$ and the third most frequent term (and) has a third as many occurrences $\text{cf}_3 = \frac{1}{3} \text{cf}_1$ etc.
Zipf’s law: The $i^{\text{th}}$ most frequent term has frequency proportional to $1/i$.

- $\text{cf}_i \propto \frac{1}{i}$
- $\text{cf}$ is collection frequency: the number of occurrences of the term in the collection.

So if the most frequent term (*the*) occurs $\text{cf}_1$ times, then the second most frequent term (*of*) has half as many occurrences $\text{cf}_2 = \frac{1}{2} \text{cf}_1$ . . .

. . . and the third most frequent term (*and*) has a third as many occurrences $\text{cf}_3 = \frac{1}{3} \text{cf}_1$ etc.

Equivalent: $\text{cf}_i = ci^k$ and $\log \text{cf}_i = \log c + k \log i$ (for $k = -1$)
Zipf’s law

- Zipf’s law: The $i^{th}$ most frequent term has frequency proportional to $1/i$.
- $cf_i \propto \frac{1}{i}$
- $cf$ is collection frequency: the number of occurrences of the term in the collection.

So if the most frequent term (the) occurs $cf_1$ times, then the second most frequent term (of) has half as many occurrences $cf_2 = \frac{1}{2} cf_1$ . . .

. . . and the third most frequent term (and) has a third as many occurrences $cf_3 = \frac{1}{3} cf_1$ etc.

- Equivalent: $cf_i = c i^k$ and $\log cf_i = \log c + k \log i$ (for $k = -1$)
- Example of a power law
### Zipf’s law for Reuters

<table>
<thead>
<tr>
<th>Recap</th>
<th>Compression</th>
<th>Term statistics</th>
<th>Dictionary compression</th>
<th>Postings compression</th>
</tr>
</thead>
</table>

Schütze: Index compression
Zipf’s law for Reuters
Zipf’s law for Reuters

![Graph showing Zipf’s law for Reuters](image-url)
Fit is not great. What is important is the key insight: Few frequent terms, many rare terms.
Dictionary compression
Dictionary compression

- The dictionary is small compared to the postings file.
Dictionary compression

- The dictionary is small compared to the postings file.
- But we want to keep it in memory.
Dictionary compression

- The dictionary is small compared to the postings file.
- But we want to keep it in memory.
- Also: competition with other applications, cell phones, onboard computers, fast startup time
The dictionary is small compared to the postings file. 

But we want to keep it in memory.

Also: competition with other applications, cell phones, onboard computers, fast startup time

So compressing the dictionary is important.
Recall: Dictionary as array of fixed-width entries
Recall: Dictionary as array of fixed-width entries

<table>
<thead>
<tr>
<th>term</th>
<th>document frequency</th>
<th>pointer to postings list</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>

Space needed: 20 bytes, 4 bytes, 4 bytes

Space for Reuters: \((20 + 4 + 4) \times 400,000 = 11.2 \text{ MB}\)
Fixed-width entries are bad.
Fixed-width entries are bad.

- Most of the bytes in the term column are wasted.
Fixed-width entries are bad.

- Most of the bytes in the term column are wasted.
  - We allot 20 bytes for terms of length 1.
Fixed-width entries are bad.

- Most of the bytes in the term column are wasted.
  - We allot 20 bytes for terms of length 1.
- We can’t handle HYDROCHLOROFLUOROCARBONS and SUPERCALIFRAGILISTICEXPIALIDOCIOUS
Fixed-width entries are bad.

- Most of the bytes in the term column are wasted.
  - We allot 20 bytes for terms of length 1.
- We can’t handle HYDROCHLOROFLUOROCARBONS and SUPERCALIFRAGILISTICEXPIALIDOCIOUS
- Average length of a term in English: 8 characters (or a little bit less)
Fixed-width entries are bad.

Most of the bytes in the term column are wasted.

- We allot 20 bytes for terms of length 1.

- We can’t handle hydrochlorofluorocarbons and supercalifragilisticexpialidocious

- Average length of a term in English: 8 characters (or a little bit less)

- How can we use on average 8 characters per term?
Dictionary as a string
Dictionary as a string

...systileszygetic\textit{icsyzygial}syzyg\textit{yszaibelyiteszecinszono}...

<table>
<thead>
<tr>
<th>freq.</th>
<th>postings ptr.</th>
<th>term ptr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>→</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

4 bytes 4 bytes 3 bytes
<table>
<thead>
<tr>
<th>Recap</th>
<th>Compression</th>
<th>Term statistics</th>
<th>Dictionary compression</th>
<th>Postings compression</th>
</tr>
</thead>
</table>

Space for dictionary as a string
Space for dictionary as a string

- 4 bytes per term for frequency
Space for dictionary as a string

- 4 bytes per term for frequency
- 4 bytes per term for pointer to postings list
Space for dictionary as a string

- 4 bytes per term for frequency
- 4 bytes per term for pointer to postings list
- 8 bytes (on average) for term in string
Space for dictionary as a string

- 4 bytes per term for frequency
- 4 bytes per term for pointer to postings list
- 8 bytes (on average) for term in string
- 3 bytes per pointer into string (need $\log_2 8 \cdot 400000 < 24$ bits to resolve $8 \cdot 400,000$ positions)
Space for dictionary as a string

- 4 bytes per term for frequency
- 4 bytes per term for pointer to postings list
- 8 bytes (on average) for term in string
- 3 bytes per pointer into string (need \(\log_{2} 8 \cdot 400000 < 24\) bits to resolve \(8 \cdot 400,000\) positions)
- Space: \(400,000 \times (4 + 4 + 3 + 8) = 7.6\)MB (compared to \(11.2\)MB for fixed-width array)
Dictionary as a string with blocking
Dictionary as a string with blocking

...7ystilestone9szygetic8szygial6szygy11szaielyite6szecin...

freq. postings ptr. term ptr.
9 →
92 →
5 →
71 →
12 →
... → ...
... →
Space for dictionary as a string with blocking
Space for dictionary as a string with blocking

- Example block size $k = 4$
Space for dictionary as a string with blocking

- Example block size $k = 4$
- Where we used $4 \times 3$ bytes for term pointers without blocking
Space for dictionary as a string with blocking

- Example block size $k = 4$
- Where we used $4 \times 3$ bytes for term pointers without blocking
  ...
- ...we now use 3 bytes for one pointer plus 4 bytes for indicating the length of each term.
Space for dictionary as a string with blocking

- Example block size \( k = 4 \)
- Where we used \( 4 \times 3 \) bytes for term pointers without blocking 
  
  \[ \ldots \]

- \( \ldots \) we now use 3 bytes for one pointer plus 4 bytes for indicating the length of each term.

- We save \( 12 - (3 + 4) = 5 \) bytes per block.
Space for dictionary as a string with blocking

- Example block size $k = 4$
- Where we used $4 \times 3$ bytes for term pointers without blocking
  ...
- ...we now use 3 bytes for one pointer plus 4 bytes for indicating the length of each term.
- We save $12 - (3 + 4) = 5$ bytes per block.
- Total savings: $400,000/4 \times 5 = 0.5 \text{ MB}$
Space for dictionary as a string with blocking

- Example block size $k = 4$
- Where we used $4 \times 3$ bytes for term pointers without blocking
  
  ... we now use 3 bytes for one pointer plus 4 bytes for indicating the length of each term.
- We save $12 - (3 + 4) = 5$ bytes per block.
- Total savings: $400,000 / 4 \times 5 = 0.5$ MB
- This reduces the size of the dictionary from 7.6 MB to 7.1 MB.
Lookup of a term without blocking
Lookup of a term with blocking: (slightly) slower
Front coding

One block in blocked compression \((k = 4)\)...

\[ \text{8 automata 8 automate 9 automatic 10 automation} \]

\[ \downarrow \]

...further compressed with front coding.

\[ \text{8 automata 1 e 2 i c 3 i o n} \]
Dictionary compression for Reuters: Summary
## Dictionary compression for Reuters: Summary

<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>
</tbody>
</table>
Exercise
Exercise

- Which prefixes should be used for front coding? What are the tradeoffs?
- Input: list of terms (= the term vocabulary)
- Output: list of prefixes that will be used in front coding
Outline

1 Recap
2 Compression
3 Term statistics
4 Dictionary compression
5 Postings compression
Postings compression
The postings file is much larger than the dictionary, factor of at least 10.
Postings compression

- The postings file is much larger than the dictionary, factor of at least 10.
- Key desideratum: store each posting compactly
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.
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.
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 \( \log_2 800,000 \approx 19.6 < 20 \) bits per docID.
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 $\log_2 800,000 \approx 19.6 < 20$ bits per docID.

Our goal: use a lot less than 20 bits per docID.
Key idea: Store gaps instead of docIDs
Key idea: Store gaps instead of docIDs

- Each postings list is ordered in increasing order of docID.
Key idea: Store gaps instead of docIDs

- Each postings list is ordered in increasing order of docID.
- Example postings list: `COMPUTER: 283154, 283159, 283202, ...`
Key idea: Store gaps instead of docIDs

- Each postings list is ordered in increasing order of docID.
- Example postings list: `COMPUTER: 283154, 283159, 283202, ...
- It suffices to store **gaps**: 283159-283154=5, 283202-283159=43
Key idea: Store gaps instead of docIDs

- Each postings list is ordered in increasing order of docID.
- Example postings list: `COMPUTER: 283154, 283159, 283202, ...
- It suffices to store gaps: 283159-283154=5, 283202-283159=43
- Example postings list using gaps: `COMPUTER: 283154, 5, 43, ...`
Key idea: Store gaps instead of docIDs

- Each postings list is ordered in increasing order of docID.
- Example postings list: **COMPUTER**: 283154, 283159, 283202, ...
- It suffices to store gaps: 283159-283154=5, 283202-283159=43
- Example postings list using gaps: **COMPUTER**: 283154, 5, 43, ...
- Gaps for frequent terms are small.
Key idea: Store gaps instead of docIDs

- Each postings list is ordered in increasing order of docID.
- Example postings list: \texttt{COMPUTER: 283154, 283159, 283202, ...}
- It suffices to store gaps: 283159-283154=5, 283202-283159=43
- Example postings list using gaps: \texttt{COMPUTER: 283154, 5, 43, ...}
- Gaps for frequent terms are small.
- Thus: We can encode small gaps with fewer than 20 bits.
### Gap encoding

<table>
<thead>
<tr>
<th>encoding</th>
<th>postings list</th>
<th>docIDs</th>
<th>gaps</th>
<th>docIDs</th>
<th>gaps</th>
<th>docIDs</th>
<th>gaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>THE</td>
<td></td>
<td>283042</td>
<td>1</td>
<td>283043</td>
<td>1</td>
<td>283044</td>
<td>1</td>
</tr>
<tr>
<td>COMPUTER</td>
<td></td>
<td>283047</td>
<td>107</td>
<td>283154</td>
<td>5</td>
<td>283159</td>
<td>43</td>
</tr>
<tr>
<td>ARACHNOCENTRIC</td>
<td></td>
<td>252000</td>
<td>252000</td>
<td>248100</td>
<td>500100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Variable length encoding
Variable length encoding

- Aim:
Variable length encoding

- **Aim:**
  - For **ARACHNOCENTRIC** and other rare terms, we will use about 20 bits per gap (= posting).
Variable length encoding

- Aim:
  - For **ARACHNOCENTRIC** and other rare terms, we will use about 20 bits per gap (＝ posting).
  - For **THE** and other very frequent terms, we will use only a few bits per gap (＝ posting).
Variable length encoding

- **Aim:**
  - For *ARACHNOCENTRIC* and other rare terms, we will use about 20 bits per gap (= posting).
  - For *THE* and other very frequent terms, we will use only a few bits per gap (= posting).

- In order to implement this, we need to devise some form of **variable length encoding**.
Variable length encoding

- **Aim:**
  - For *ARACHNOCENTRIC* and other rare terms, we will use about 20 bits per gap (= posting).
  - For *THE* and other very frequent terms, we will use only a few bits per gap (= posting).

- In order to implement this, we need to devise some form of **variable length encoding**.

- Variable length encoding uses few bits for small gaps and many bits for large gaps.
Variable byte (VB) code
Variable byte (VB) code

- Used by many commercial/research systems
Variable byte (VB) code

- Used by many commercial/research systems
- Good low-tech blend of variable-length coding and sensitivity to alignment matches (bit-level codes, see later).
Variable byte (VB) code

- Used by many commercial/research systems
- Good low-tech blend of variable-length coding and sensitivity to alignment matches (bit-level codes, see later).
- Dedicate 1 bit (high bit) to be a continuation bit $c$. 
Variable byte (VB) code

- Used by many commercial/research systems
- Good low-tech blend of variable-length coding and sensitivity to alignment matches (bit-level codes, see later).
- Dedicate 1 bit (high bit) to be a continuation bit $c$.
- If the gap $G$ fits within 7 bits, binary-encode it in the 7 available bits and set $c = 1$. 
Variable byte (VB) code

- Used by many commercial/research systems
- Good low-tech blend of variable-length coding and sensitivity to alignment matches (bit-level codes, see later).
- Dedicate 1 bit (high bit) to be a continuation bit \( c \).
- If the gap \( G \) fits within 7 bits, binary-encode it in the 7 available bits and set \( c = 1 \).
- Else: encode lower-order 7 bits and then use one or more additional bytes to encode the higher order bits using the same algorithm.
Variable byte (VB) code

- Used by many commercial/research systems
- Good low-tech blend of variable-length coding and sensitivity to alignment matches (bit-level codes, see later).
- Dedicate 1 bit (high bit) to be a continuation bit $c$.
- If the gap $G$ fits within 7 bits, binary-encode it in the 7 available bits and set $c = 1$.
- Else: encode lower-order 7 bits and then use one or more 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 of the other bytes to 0 ($c = 0$).
## VB code examples

<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>10111000</td>
<td>10000101</td>
</tr>
</tbody>
</table>
**VB code encoding algorithm**

**VBEncodeNumber**\( (n) \)

1. \( \textit{bytes} \leftarrow \langle \rangle \)
2. \textbf{while} true
3. \textbf{do} \quad \texttt{Prepend}(\textit{bytes}, n \mod 128)
4. \quad \textbf{if} \ n < 128
5. \quad \quad \textbf{then} \quad \textbf{break}
6. \quad \quad \textit{n} \leftarrow \textit{n} \div 128
7. \quad \textit{bytes}[\texttt{Length}(\textit{bytes})] \textit{+=} 128
8. \textbf{return} \quad \textit{bytes}

**VBEncode**\( (\textit{numbers}) \)

1. \( \textit{bytestream} \leftarrow \langle \rangle \)
2. \textbf{for each} \ n \in \textit{numbers}
3. \quad \textit{bytes} \leftarrow \texttt{VBEncodeNumber}(n)
4. \quad \textit{bytestream} \leftarrow \texttt{Extend}(\textit{bytestream}, \textit{bytes})
5. \textbf{return} \quad \textit{bytestream}
VB code decoding algorithm

\[
\text{VBDecode}(\text{bytestream})
\]

1. \(\text{numbers} \leftarrow \langle \rangle\)
2. \(n \leftarrow 0\)
3. \(\text{for } i \leftarrow 1 \text{ to } \text{LENGTH}(\text{bytestream})\)
4. \(\text{do if } \text{bytestream}[i] < 128\)
5. \(\text{then } n \leftarrow 128 \times n + \text{bytestream}[i]\)
6. \(\text{else } n \leftarrow 128 \times n + (\text{bytestream}[i] - 128)\)
7. \(\text{APPEND(numbers, n)}\)
8. \(n \leftarrow 0\)
9. \(\text{return } \text{numbers}\)
Other variable codes
Other variable codes

- Instead of bytes, we can also use a different “unit of alignment”: 32 bits (words), 16 bits, 4 bits (nibbles) etc
Other variable codes

- Instead of bytes, we can also use a different “unit of alignment”: 32 bits (words), 16 bits, 4 bits (nibbles) etc
- Variable byte alignment wastes space if you have many small gaps – nibbles do better on those.
Other variable codes

- Instead of bytes, we can also use a different “unit of alignment”: 32 bits (words), 16 bits, 4 bits (nibbles) etc
- Variable byte alignment wastes space if you have many small gaps – nibbles do better on those.
- There is work on word-aligned codes that efficiently “pack” a variable number of gaps into one word – see resources at the end
Gamma codes for gap encoding
You can get even more compression with another type of variable length encoding: \textit{bitlevel} code.
Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: `bitlevel` code.
- Gamma code is the best known of these.
Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: \textit{bitlevel} code.
- Gamma code is the best known of these.
- First, we need unary code to be able to introduce gamma code.
Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: \textit{bitlevel} code.
- Gamma code is the best known of these.
- First, we need unary code to be able to introduce gamma code.
- Unary code
You can get even more compression with another type of variable length encoding: \textit{bitlevel} code.

Gamma code is the best known of these.

First, we need unary code to be able to introduce gamma code.

Unary code
  - Represent $n$ as $n$ 1s with a final 0.
Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: **bitlevel** code.
- Gamma code is the best known of these.
- First, we need unary code to be able to introduce gamma code.
- **Unary code**
  - Represent $n$ as $n$ 1s with a final 0.
  - Unary code for 3 is 1110
Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: bitlevel code.
- Gamma code is the best known of these.
- First, we need unary code to be able to introduce gamma code.

Unary code
- Represent $n$ as $n$ 1s with a final 0.
- Unary code for 3 is 1110
- Unary code for 40 is
  
  1111111111111111111111111111111111111111111111110
Gamma codes for gap encoding

- You can get even more compression with another type of variable length encoding: **bitlevel** code.
- Gamma code is the best known of these.
- First, we need unary code to be able to introduce gamma code.

**Unary code**

- Represent \( n \) as \( n \) 1s with a final 0.
- Unary code for 3 is 1110
- Unary code for 40 is 
  
  111111111111111111111111111111111111111111111111111111111111111111111110

- Unary code for 70 is:

  111111111111111111111111111111111111111111111111111111111111111111111110
Gamma code
Gamma code

- Represent a gap $G$ as a pair of **length** and **offset**.
Gamma code

- Represent a gap $G$ as a pair of **length** and **offset**.
- Offset is the gap in binary, with the leading bit chopped off.
Gamma code

- Represent a gap $G$ as a pair of length and offset.
- Offset is the gap in binary, with the leading bit chopped off.
- For example $13 \rightarrow 1101 \rightarrow 101 = \text{offset}$
Gamma code

- Represent a gap $G$ as a pair of **length** and **offset**.
- Offset is the gap in binary, with the leading bit chopped off.
- For example $13 \rightarrow 1101 \rightarrow 101 = \text{offset}$
- Length is the length of offset.
Gamma code

- Represent a gap $G$ as a pair of length and offset.
- Offset is the gap in binary, with the leading bit chopped off.
- For example $13 \rightarrow 1101 \rightarrow 101 = \text{offset}$
- Length is the length of offset.
- For $13$ (offset $101$), this is $3$. 
Gamma code

- Represent a gap $G$ as a pair of length and offset.
- Offset is the gap in binary, with the leading bit chopped off.
- For example $13 \rightarrow 1101 \rightarrow 101 = \text{offset}$
- Length is the length of offset.
- For 13 (offset 101), this is 3.
- Encode length in unary code: 1110.
Represent a gap $G$ as a pair of length and offset.
Offset is the gap in binary, with the leading bit chopped off.
For example $13 \rightarrow 1101 \rightarrow 101 = \text{offset}$
Length is the length of offset.
For $13$ (offset $101$), this is $3$.
Encode length in 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>unary code</th>
<th>length</th>
<th>offset</th>
<th>( \gamma ) code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>10,0</td>
</tr>
<tr>
<td>2</td>
<td>110</td>
<td>10</td>
<td>0</td>
<td>10,1</td>
</tr>
<tr>
<td>3</td>
<td>1110</td>
<td>10</td>
<td>1</td>
<td>110,00</td>
</tr>
<tr>
<td>4</td>
<td>11110</td>
<td>110</td>
<td>00</td>
<td>1110,001</td>
</tr>
<tr>
<td>9</td>
<td>1111111110</td>
<td>1110</td>
<td>001</td>
<td>1110,101</td>
</tr>
<tr>
<td>13</td>
<td>1110</td>
<td>101</td>
<td></td>
<td>11110,1000</td>
</tr>
<tr>
<td>24</td>
<td>11110</td>
<td>1000</td>
<td>0000000001</td>
<td>1111111110,1111111111</td>
</tr>
<tr>
<td>511</td>
<td>111111110</td>
<td>11111111</td>
<td>111111110,111111111</td>
<td>11111111110,0000000001</td>
</tr>
<tr>
<td>1025</td>
<td>1111111110</td>
<td>00000000001</td>
<td>11111111110,0000000001</td>
<td></td>
</tr>
</tbody>
</table>
Exercise

- Compute the variable byte code of 130
- Compute the gamma code of 130
Length of gamma code
The length of offset is \( \lfloor \log_2 G \rfloor \) bits.
Length of gamma code

- The length of offset is $\lfloor \log_2 G \rfloor$ bits.
- The length of length is $\lfloor \log_2 G \rfloor + 1$ bits,
Length of gamma code

- The length of offset is $\lfloor \log_2 G \rfloor$ bits.
- The length of length is $\lfloor \log_2 G \rfloor + 1$ bits,
- So the length of the entire code is $2 \times \lfloor \log_2 G \rfloor + 1$ bits.
Length of gamma code

- The length of offset is $\lfloor \log_2 G \rfloor$ bits.
- The length of length is $\lfloor \log_2 G \rfloor + 1$ bits,
- So the length of the entire code is $2 \times \lfloor \log_2 G \rfloor + 1$ bits.
- $\gamma$ codes are always of odd length.
The length of offset is \( \lceil \log_2 G \rceil \) bits.

The length of length is \( \lceil \log_2 G \rceil + 1 \) bits,

So the length of the entire code is \( 2 \times \lceil \log_2 G \rceil + 1 \) bits.

\( \gamma \) codes are always of odd length.

Gamma codes are within a factor of 2 of the optimal encoding length \( \log_2 G \).
Length of gamma code

- The length of *offset* is $\lfloor \log_2 G \rfloor$ bits.
- The length of *length* is $\lfloor \log_2 G \rfloor + 1$ bits,
- So the length of the entire code is $2 \times \lfloor \log_2 G \rfloor + 1$ bits.
- $\gamma$ codes are always of odd length.
- Gamma codes are within a factor of 2 of the optimal encoding length $\log_2 G$.
  - (assuming the frequency of a gap $G$ is proportional to $\log_2 G$ – only approximately true)
Gamma code: Properties
Gamma code: Properties

- Gamma code (like variable byte code) is **prefix-free**: a valid code word is not a prefix of any other valid code.
Gamma code: Properties

- Gamma code (like variable byte code) is **prefix-free**: a valid code word is not a prefix of any other valid code.
- Encoding is optimal within a factor of 3 (and within a factor of 2 making additional assumptions).
Gamma code: Properties

- Gamma code (like variable byte code) is **prefix-free**: a valid code word is not a prefix of any other valid code.
- Encoding is optimal within a factor of 3 (and within a factor of 2 making additional assumptions).
- This result is independent of the distribution of gaps!
Gamma code: Properties

- Gamma code (like variable byte code) is prefix-free: a valid code word is not a prefix of any other valid code.
- Encoding is optimal within a factor of 3 (and within a factor of 2 making additional assumptions).
- This result is independent of the distribution of gaps!
- We can use gamma codes for any distribution. Gamma code is universal.
Gamma code: Properties

- Gamma code (like variable byte code) is prefix-free: a valid code word is not a prefix of any other valid code.
- Encoding is optimal within a factor of 3 (and within a factor of 2 making additional assumptions).
- This result is independent of the distribution of gaps!
- We can use gamma codes for any distribution. Gamma code is universal.
- Gamma code is parameter-free.
Gamma codes: Alignment
Gamma codes: Alignment

- Machines have word boundaries – 8, 16, 32 bits
Gamma codes: Alignment

- Machines have word boundaries – 8, 16, 32 bits
- Compressing and manipulating at granularity of bits can be slow.
Gamma codes: Alignment

- Machines have word boundaries – 8, 16, 32 bits
- Compressing and manipulating at granularity of bits can be slow.
- Variable byte encoding is aligned and thus potentially more efficient.
Gamma codes: Alignment

- Machines have word boundaries – 8, 16, 32 bits
- Compressing and manipulating at 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.
## Compression of Reuters

<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>3600.0</td>
</tr>
<tr>
<td>Collection (text)</td>
<td>960.0</td>
</tr>
<tr>
<td>T/D 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>
## Term-document incidence matrix

<table>
<thead>
<tr>
<th></th>
<th>Anthony and Cleopatra</th>
<th>Julius Caesar</th>
<th>The Tempest</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Anthony</strong></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>Brutus</strong></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Caesar</strong></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Calpurnia</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Cleopatra</strong></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Mercy</strong></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Worse</strong></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Entry is 1 if term occurs. Example: **Calpurnia** occurs in *Julius Caesar*.

Entry is 0 if term doesn’t occur. Example: **Calpurnia** doesn’t occur in *The tempest*. 
## Compression of Reuters

<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>3600.0</td>
</tr>
<tr>
<td>Collection (text)</td>
<td>960.0</td>
</tr>
<tr>
<td>T/D 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>
## Compression of Reuters

<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>3600.0</td>
</tr>
<tr>
<td>Collection (text)</td>
<td>960.0</td>
</tr>
<tr>
<td>T/D 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>
Summary
We can now create an index for highly efficient Boolean retrieval that is very space efficient.
Summary

- We can now create an index for highly efficient Boolean retrieval that is very space efficient.
- Only 10-15% of the total size of the text in the collection.
Summary

- We can now create an index for highly efficient Boolean retrieval that is very space efficient.
- Only 10-15% of the total size of the text in the collection.
- However, we’ve ignored positional and frequency information.
Summary

- We can now create an index for highly efficient Boolean retrieval that is very space efficient.
- Only 10-15% of the total size of the text in the collection.
- However, we’ve ignored positional and frequency information.
- For this reason, space savings are less in reality.
Take-away today

- Motivation for compression in information retrieval systems
- How can we compress the dictionary component of the inverted index?
- How can we compress the postings component of the inverted index?
- Term statistics: how are terms distributed in document collections?
Resources

- Chapter 5 of IIR
- Resources at http://cislmu.org
  - Original publication on word-aligned binary codes by Anh and Moffat (2005); also: Anh and Moffat (2006a)
  - Original publication on variable byte codes by Scholer, Williams, Yiannis and Zobel (2002)
  - More details on compression (including compression of positions and frequencies) in Zobel and Moffat (2006)