Lecture 2: Data structures and Algorithms for Indexing
Information Retrieval
Computer Science Tripos Part II

Ronan Cummins

Natural Language and Information Processing (NLIP) Group
UNIVERSITY OF CAMBRIDGE
ronan.cummins@cl.cam.ac.uk

2017

1Adapted from Simone Teufel’s original slides
IR System Components

Query → IR System → Set of relevant documents

Document Collection
IR System Components

- **Query**
- **Document Collection**
- **Document Normalisation**
- **Indexer**
- **Ranking/Matching Module**
- **Set of relevant documents**
- **Indexes**

Today: The indexer
Today: The indexer
Overview

1. Index construction
   - Postings list and Skip lists
   - Single-pass Indexing

2. Document and Term Normalisation
   - Documents
   - Terms
   - Reuter RCV1 and Heap’s Law
The major steps in inverted index construction:

- Collect the documents to be indexed.
- Tokenize the text.
- Perform linguistic preprocessing of tokens.
- Index the documents that each term occurs in.
Example: index creation by sorting

Table 1: Term document (docID)

<table>
<thead>
<tr>
<th>Term</th>
<th>docID</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>i'</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
</tr>
<tr>
<td>so</td>
<td>2</td>
</tr>
<tr>
<td>let</td>
<td>2</td>
</tr>
<tr>
<td>it</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
</tr>
<tr>
<td>with</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
</tr>
<tr>
<td>noble</td>
<td>2</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
</tr>
<tr>
<td>hath</td>
<td>2</td>
</tr>
<tr>
<td>told</td>
<td>2</td>
</tr>
<tr>
<td>you</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
</tr>
<tr>
<td>ambitious</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 2: Term (sorted) document

<table>
<thead>
<tr>
<th>Term (sorted)</th>
<th>docID</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambitious</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
</tr>
<tr>
<td>brutus</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
</tr>
<tr>
<td>capitol</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
</tr>
<tr>
<td>hath</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>i'</td>
<td>1</td>
</tr>
<tr>
<td>it</td>
<td>2</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>2</td>
</tr>
<tr>
<td>let</td>
<td>2</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
</tr>
<tr>
<td>noble</td>
<td>2</td>
</tr>
<tr>
<td>so</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
</tr>
<tr>
<td>told</td>
<td>2</td>
</tr>
<tr>
<td>you</td>
<td>2</td>
</tr>
<tr>
<td>was</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>1</td>
</tr>
<tr>
<td>with</td>
<td>2</td>
</tr>
</tbody>
</table>

Doc 1:
I did enact Julius
Caesar: I was killed
i' the Capitol; Brutus killed me.

Doc 2:
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious.
Index creation; grouping step ("uniq")

<table>
<thead>
<tr>
<th>Term</th>
<th>Doc. freq.</th>
<th>Postings list</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambitious</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>hath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>i’</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>it</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>let</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>me</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>noble</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>so</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>the</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>told</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>you</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>was</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>with</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

- Primary sort by term (dictionary)
- Secondary sort (within postings list) by document ID
- Document frequency (length of postings list):
  - for more efficient Boolean searching (later today)
  - for term weighting (lecture 4)
- keep Dictionary in memory
- keep Postings List (much larger) on disk
Data structures for Postings Lists

- **Singly linked list**
  - Allow cheap insertion of documents into postings lists (e.g., when recrawling)
  - Naturally extend to skip lists for faster access

- **Variable length array**
  - Better in terms of space requirements
  - Also better in terms of time requirements if memory caches are used, as they use contiguous memory

- **Hybrid scheme**: linked list of variable length array for each term.
  - write posting lists on disk as contiguous block without explicit pointers
  - minimises the size of postings lists and number of disk seeks
Some postings lists can contain several million entries
Check skip list if present to skip multiple entries
\[ \sqrt{L} \] Skips can be placed evenly for a list of length \( L \).
Tradeoff Skip Lists

Number of items skipped vs. frequency that skip can be taken

- More skips: each pointer skips only a few items, but we can frequently use it.
- Fewer skips: each skip pointer skips many items, but we can not use it very often.
- Skip pointers used to help a lot, but with today’s fast CPUs, they don’t help that much anymore.
As we build index, we parse docs one at a time.
The final postings for any term are incomplete until the end.
But for large collections, we cannot keep all postings in memory and then sort in-memory at the end.
We cannot sort very large sets of records on disk either (too many disk seeks, expensive).
Thus: We need to store intermediate results on disk.
We need a scalable **Block-Based** sorting algorithm.
Abbreviation: SPIMI

Key idea 1: Generate separate dictionaries for each block.

Key idea 2: 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.

Worked example!
Single-pass in-memory indexing (2)

postings lists to be merged

<table>
<thead>
<tr>
<th>brutus</th>
<th>d1,d3</th>
</tr>
</thead>
<tbody>
<tr>
<td>caesar</td>
<td>d1,d2,d4</td>
</tr>
<tr>
<td>noble</td>
<td>d5</td>
</tr>
<tr>
<td>with</td>
<td>d1,d2,d3,d5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>brutus</th>
<th>d6,d7</th>
</tr>
</thead>
<tbody>
<tr>
<td>caesar</td>
<td>d8,d9</td>
</tr>
<tr>
<td>julius</td>
<td>d10</td>
</tr>
<tr>
<td>killed</td>
<td>d8</td>
</tr>
<tr>
<td>noble</td>
<td>d5</td>
</tr>
<tr>
<td>with</td>
<td>d1,d2,d3,d5</td>
</tr>
</tbody>
</table>

merged postings lists

<table>
<thead>
<tr>
<th>brutus</th>
<th>d1,d3,d6,d7</th>
</tr>
</thead>
<tbody>
<tr>
<td>caesar</td>
<td>d1,d2,d4,d8,d9</td>
</tr>
<tr>
<td>julius</td>
<td>d10</td>
</tr>
<tr>
<td>killed</td>
<td>d8</td>
</tr>
<tr>
<td>noble</td>
<td>d5</td>
</tr>
<tr>
<td>with</td>
<td>d1,d2,d3,d5</td>
</tr>
</tbody>
</table>

disk
We could save space in memory by assigning term-ids to terms for each block-based dictionary. However, we then need to have an in-memory term-term-id mapping which often does not fit in memory (on a single machine at least).

This approach is called *blocked sort-based indexing* BSBI and you can read about it in the book (Chapter 4.2).
Overview

1. Index construction
   - Postings list and Skip lists
   - Single-pass Indexing

2. Document and Term Normalisation
   - Documents
   - Terms
   - Reuter RCV1 and Heap’s Law
To build an inverted index, we need to get from

- Input: Friends, Romans, countrymen. So let it be with Caesar...
- Output: friend, roman, countryman, so
- Each token is a candidate for a postings entry.
- What are valid tokens to emit?
Up to now, we assumed that
- We know what a document is.
- We can “machine-read” each document

More complex in reality
We need do deal with format and language of each document
Format could be excel, pdf, latex, word...
What language is it in?
What character set is it in?
Each of these is a statistical classification problem
Alternatively we can use heuristics
Text is not just a linear stream of logical “characters”...

- Determine correct character encoding (Unicode UTF-8) – by ML or by metadata or heuristics.
- Compressions, binary representation (DOC)
- Treat XML characters separately (amp)
A single index usually contains terms of several languages.

Documents or their components can contain multiple languages/format, for instance a French email with a Spanish pdf attachment.

What is the document unit for indexing?
- a file?
- an email?
- an email with 5 attachments?
- an email thread?

Answering the question “What is a document?” is not trivial.

Smaller units raise precision, drop recall.

Also might have to deal with XML/hierarchies of HTML documents etc.
Normalisation

- Need to normalise words in the indexed text as well as query terms to the same form.
- Example: We want to match U.S.A. to USA.
- We most commonly implicitly define equivalence classes of terms.
- Alternatively, we could do asymmetric expansion:
  
  window $\rightarrow$ window, windows  
  windows $\rightarrow$ Windows, windows, window  
  Windows $\rightarrow$ Windows

- Either at query time, or at index time.
- More powerful, but less efficient.
Mr. O’Neill thinks that the boys’ stories about Chile’s capital aren’t amusing.
Tokenisation problems: One word or two? (or several)

- Hewlett-Packard
- State-of-the-art
- co-education
- the hold-him-back-and-drag-him-away maneuver
- data base
- San Francisco
- Los Angeles-based company
- cheap San Francisco-Los Angeles fares
- York University vs. New York University
Older IR systems may not index numbers...

... but generally it’s a useful feature.
莎拉波娃现在居住在美国东南部的佛罗里达。今年4月9日，莎拉波娃在美国第一大城市纽约度过了18岁生日。生日派对上，莎拉波娃露出了甜美的微笑。

- Need to perform word segmentation
- Use a lexicon or supervised machine-learning
和尚

- As one word, means “monk”
- As two words, means “and” and “still”
Compounding in Dutch, German, Swedish

<table>
<thead>
<tr>
<th>German</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lebensversicherungsgesellschaftsangestellter</td>
</tr>
<tr>
<td>leben+versicherung+gesellschaft+angestellter</td>
</tr>
</tbody>
</table>
Other cases of “no whitespace”: Agglutination

“Agglutinative” languages do this not just for compounds:

**Inuit**

tusaatsiarunngittualuujunga
(= “I can’t hear very well”)

**Finnish**

epäjärjestelmällistytämättömyydellänsäkäänköhän
(= “I wonder if – even with his/her quality of not having been made unsystematized”)

**Turkish**

Çekoslovakyalılaştıramadıklarımızdanmışçasına
(= “as if you were one of those whom we could not make resemble the Czechoslovakian people”)

Different scripts (alphabets) might be mixed in one language.

Japanese has 4 scripts: kanja, katakana, hiragana, Romanji

no spaces
Direction of writing changes in some scripts (writing systems); e.g., Arabic.

‘Algeria achieved its independence in 1962 after 132 years of French occupation.’

Rendering vs. conceptual order

Bidirectionality is not a problem if Unicode encoding is chosen
Accents and diacritics

- résumé vs. resume
- Universität
- Meaning-changing in some languages:

  peña = cliff, pena = sorrow
  (Spanish)

- Main questions: will users apply it when querying?
Case Folding

- Reduce all letters to lower case
- Even though case can be semantically distinguishing
  - Fed vs. fed
  - March vs. march
  - Turkey vs. turkey
  - US vs. us
- Best to reduce to lowercase because users will use lowercase regardless of correct capitalisation.
Stop words

- Extremely common words which are of little value in helping select documents matching a user need
  - a, an, and, are, as, at, be, by, for, from, has, he, in, is, it, its, of, on, that, the, to, was, were, will, with

- Used to be standard in older IR systems.
- Need them to search for

  to be or not to be prince of Denmark bamboo in water

- Length of practically used stoplists has shrunk over the years.
- Most web search engines do index stop words.
More equivalence classing

- Thesauri: semantic equivalence, car = automobile
- Soundex: phonetic equivalence, Muller = Mueller; lecture 3
Lemmatisation

- Reduce inflectional/variant forms to base form

  am, are, is → be
  car, car’s, cars’, cars → car
  the boy’s cars are different colours → the boy car be different color

- Lemmatisation implies doing “proper” reduction to dictionary headword form (the lemma)

- Inflectional morphology (cutting → cut)

- Derivational morphology (destruction → destroy)
Stemming is a crude heuristic process that **chops off the ends of words** in the hope of achieving what “principled” lemmatisation attempts to do with a lot of linguistic knowledge.

- **language dependent,** but fast and space-efficient
- **does not require** a stem dictionary, only a suffix dictionary
- **Often both inflectional and derivational**

automate, automation, automatic  $\rightarrow$ automat

- Root changes (deceive/deception, resume/resumption) aren’t dealt with, but these are rare

Most common algorithm for stemming English

Results suggest it is at least as good as other stemmers

Syllable-like shapes + 5 phases of reductions

Of the rules in a compound command, select the top one and exit that compound (this rule will have affecte the longest suffix possible, due to the ordering of the rules).
Stemming: Representation of a word

\[
[C] (VC)\{m\}[V]
\]

- **C**: one or more adjacent consonants
- **V**: one or more adjacent vowels
- **[ ]**: optionality
- **( )**: group operator
- **\{x\}**: repetition \(x\) times
- **\(m\)**: the “measure” of a word

Notation: measure \(m\) is calculated on the word **excluding** the suffix of the rule under consideration

- **shoe**: \([sh]_C[oe]_V\) \(m=0\)
- **Mississippi**: \([M]_C([i]_V[ss]_C)([i]_V[ss]_C)([i]_V[pp]_C)[i]_V\) \(m=3\)
- **ears**: \(([ea]_V[rs]_C)\) \(m=1\)
Porter stemmer: selected rules

SSES $\rightarrow$ SS
IES $\rightarrow$ I
SS $\rightarrow$ SS
S $\rightarrow$

caresse $\rightarrow$ caress
cares $\rightarrow$ care

$(m>0)$ EED $\rightarrow$
EE

feed $\rightarrow$ feed
agreed $\rightarrow$ agree
BUT: freed, succeed
Porter Stemmer: selected rules

\[ (*v*) \text{ ED } \rightarrow \]

plastered $\rightarrow$ plaster
bled $\rightarrow$ bled
Three stemmers: a comparison

Such an analysis can reveal features that are not easily visible from the variations in the individual genes and can lead to a picture of expression that is more biologically transparent and accessible to interpretation.

### Porter Stemmer

| such an analysi can reveal featur that ar not easili visibl from the variat in the individu gene and can lead to a pictur of express that is more biolog transpar and access to interpret |

### Lovins Stemmer

| such an analys can reve featur that ar not eas vis from th vari in th individu gen and can lead to a pictur of expres that is mor biolog transpar and acces to interpres |

### Paice Stemmer

| such an analys can rev feat that are not easy vis from the vary in the individ gen and can lead to a pict of express that is mor biolog transp and access to interpret |
In general, stemming increases effectiveness for some queries and decreases it for others.

**Example queries where stemming helps**

- `tartan sweaters` → sweater, sweaters
- `sightseeing tour san francisco` → tour, tours

**Example queries where stemming hurts**

- `operational research` → “oper” = operates, operatives, operate, operation, operational, operative
- `operating system` → operates, operatives, operate, operation, operational, operative
- `operative dentistry` → operates, operatives, operate, operation, operational, operative
We want to answer a query such as `[cambridge university]` – as a phrase.

The Duke of Cambridge recently went for a term-long course to a famous university should not be a match.

About 10% of web queries are phrase queries.

Consequence for inverted indexes: no longer sufficient to store docIDs in postings lists.

Two ways of extending the inverted index:
- biword index
- positional index
Index every consecutive pair of terms in the text as a phrase.

Example:

Friends, Romans, Countrymen

Generates two biwords:

- friends romans
- romans countrymen

Each of these biwords is now a vocabulary term.

Two-word phrases can now easily be answered.
A long phrase like *cambridge university west campus* can be represented as the Boolean query

\[
\text{cambridge university AND university west AND west campus}
\]

We need to do post-filtering of hits to identify subset that actually contains the 4-word phrase.
Issues with biword indexes

- Why are biword indexes rarely used?
- False positives, as noted above
- Index blowup due to very large term vocabulary
Positional indexes are a more efficient alternative to biword indexes.

Postings lists in a nonpositional index: each posting is just a docID

Postings lists in a positional index: each posting is a docID and a list of positions (offsets)
Query: “to\textsubscript{1} be\textsubscript{2} or\textsubscript{3} not\textsubscript{4} to\textsubscript{5} be\textsubscript{6}”

\begin{itemize}
    \item to, 993427:
    \begin{itemize}
        \item < 1: < 7, 18, 33, 72, 86, 231>;
        \item 2: <1, 17, 74, 222, 255>;
        \item 4: <8, 16, 190, 429, 433>;
        \item 5: <363, 367>;
        \item 7: <13, 23, 191>;
        \item \ldots \ldots >
    \end{itemize}

    \item be, 178239:
    \begin{itemize}
        \item < 1: < 17, 25>;
        \item 4: < 17, 191, 291, 430, 434>;
        \item 5: <14, 19, 101>;
        \item \ldots \ldots >
    \end{itemize}
\end{itemize}

Document 4 is a match.
(As always: docid, term, doc freq; new: offsets)
Proximity search

- We just saw how to use a positional index for phrase searches.
- We can also use it for proximity search.

**employment /4 place**

- Find all documents that contain *employment* and *place* within 4 words of each other.
- HIT: Employment agencies that place healthcare workers are seeing growth.
- NO HIT: Employment agencies that have learned to adapt now place healthcare workers.
Use the positional index

Simplest algorithm: look at cross-product of positions of (i) "employment" in document and (ii) "place" in document

Very inefficient for frequent words, especially stop words

Note that we want to return the actual matching positions, not just a list of documents.

This is important for dynamic summaries etc.
Proximity intersection

PositionalIntersect(p1, p2, k)
1 answer ← <>
2 while p1 6= nil and p2 6= nil
3 do if docID(p1) = docID(p2)
4 then l ← <>
5 pp1 ← positions(p1)
6 pp2 ← positions(p2)
7 while pp1 6= nil
8 do while pp2 6= nil
9 do if |pos(pp1) pos(pp2)| ≤ k
10 then Add(l , pos(pp2))
11 else if pos(pp2) > pos(pp1)
12 then break
13 pp2 ← next(pp2)
14 while l 6= <> and |l [0] pos(pp1)| > k
15 do Delete(l [0])
16 for each ps l
17 do Add(answer , hdocID(p1), pos(pp1), psi)
18 pp1 ← next(pp1)
19 p1 ← next(p1)
20 p2 ← next(p2)
21 else if docID(p1) < docID(p2)
22 then p1 ← next(p1)
23 else p2 ← next(p2)
24 return answer
Combination scheme

- Biword indexes and positional indexes can be profitably combined.

Many biwords are extremely frequent: Michael Jackson, Britney Spears etc

For these biwords, increased speed compared to positional postings intersection is substantial.

Combination scheme: Include frequent biwords as vocabulary terms in the index. Do all other phrases by positional intersection.

Williams et al. (2004) evaluate a more sophisticated mixed indexing scheme. Faster than a positional index, at a cost of 26% more space for index.

For web search engines, positional queries are much more expensive than regular Boolean queries.
Shakespeare’s collected works are not large enough to demonstrate scalable index construction algorithms.

Instead, we will use the Reuters RCV1 collection.

English newswire articles published in a 12 month period (1995/6)

\[
\begin{array}{|c|c|}
\hline
N & \text{documents} & 800,000 \\
M & \text{terms (}=\text{word types}) & 400,000 \\
T & \text{non-positional postings} & 100,000,000 \\
\hline
\end{array}
\]
## 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>
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
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

- 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.
Take-away

- Understanding of the basic unit of classical information retrieval systems: words and documents: What is a document, what is a term?
- Tokenization: how to get from raw text to terms (or tokens)
- More complex indexes for phrases
- MRS Chapter 2.2
- MRS Chapter 2.4
- MRS Chapter 4.3