IR - INDEXING

Jian-Yun Nie
(based on the lectures of Manning and Raghavan)
Roadmap

• Last lecture: Overview of IR
• From this lecture on, more details
• Today: Indexing process
  • Basic operations
  • Means to speedup
  • Simple linguistic processing
  • Supporting proximity search
• References
• Questions
Indexing-based IR

Document

indexing

Query

indexing

(Query analysis)

Representation (keywords)

Query evaluation, Retrieval, (Document ranking)

Representation (keywords)
Basics - Goals

• Recognize the contents of a document
  • What are the main topics of the document?
  • What are the basic units (indexing units) to represent them?
  • How to weight their importance?

• Support fast search given a query
  • Given a query, find the documents that contain the words.
Basics - document

• Document: anything which one may search for, which contains information in different media (text, image, …)
  • This course: text

• Text document = description in a natural language

• Human vs. computer understanding
  • Read the text and understand the meaning
  • A computer cannot (yet) understand meaning as a human being, but can quickly process symbols (strings, words, …).

• Indexing process:
  • Let a computer “read” a text
  • Select the symbols to represent what it believes to be important
  • Create a representation (index) in order to support fast search
Basics – “Reading” a document

• Let us use words as the representation units

• Document parsing
  • Identify document format (text, Word, PDF, …)
  • Identify different text parts (title, text body, …) (note: often separate index for different parts)
  • Go through a text, and recognize the words
    • Tokenization
      • The elements recognized = tokens
    • Statistics for weighting

• Create index structures

• Search: Query words → corresponding documents
Basic indexing pipeline

Documents to be indexed.

Token stream.

Linguistic modules

Modified tokens.

Indexer

Inverted index.

Friends, Romans, countrymen.

friend

roman

countryman

Documents to be indexed.

Token stream.

Linguistic modules

Modified tokens.

Indexer

Inverted index.

Friends, Romans, countrymen.
Tokenization

- **Input:** “Friends, Romans and Countrymen”
- **Output:** Tokens
  - *Friends*
  - *Romans*
  - *and*
  - *Countrymen*

- Usually use space and punctuations
- Each such token is now a candidate for an index entry, after further processing
Tokenization: issues

- *Finland’s capital* →
  *Finland? Finlands? Finland’s?*

- *Hewlett-Packard* →
  *Hewlett* and *Packard* as two tokens?
  - *State-of-the-art*: break up hyphenated sequence.
  - co-education?
  - the hold-him-back-and-drag-him-away-maneuver?

- *San Francisco*: one token or two? How do you decide it is one token?
Tokenization: Numbers

- 3/12/91
- Mar. 12, 1991
- 55 B.C.
- B-52
- My PGP key is 324a3df234cb23e
- 100.2.86.144
  - Generally, don’t index as text.
  - Will often index “meta-data” separately
    - Creation date, format, etc.
Tokenization: Language issues

• **L'ensemble** (the set) → one token or two?
  • *L* ? *L ’* ? *Le* ?
  • Want **ensemble** to match with **un ensemble**
  • but how about **aujourd’hui** (today)
  • → create a dictionary to store special words such as **aujourd’hui**

• German noun compounds are not segmented
  • Lebensversicherungsgesellschaftsangestellter
  • ‘life insurance company employee’
  • → resort to a post- processing of decompounding
Tokenization: language issues

- Chinese and Japanese have no spaces between words:
  - Not always guaranteed a unique tokenization
- Further complicated in Japanese, with multiple alphabets intermingled
  - Dates/amounts in multiple formats

フォーチュン500社は情報不足のため時間あたり$500K(約6,000万円)

Katakana  Hiragana  Kanji  “Romaji”

End-user can express query entirely in hiragana!
Normalization

• Need to “normalize” terms in indexed text as well as query terms into the same form
  • We want to match U.S.A. and USA
• We most commonly implicitly define equivalence classes of terms
  • e.g., by deleting periods in a term
  • U.S. → US
Case folding

- Reduce all letters to lower case
  - exception: upper case (in mid-sentence?)
    - e.g., *General Motors*
    - *Fed* vs. *fed*
    - *SAIL* vs. *sail*
    - *AIDS* vs. *aids*

- Often best to lower case everything, since users will use lowercase regardless of ‘correct’ capitalization

- Problems
  - Simple tokenization: U.S. → U, S
  - U.S. → us

- **Question**: Other problems in tokenization?
A naïve representation of indexing result: Term-document incidence

<table>
<thead>
<tr>
<th>Terms</th>
<th>Antony and Cleopatra</th>
<th>Julius Caesar</th>
<th>The Tempest</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Brutus</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>worser</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

1 if document contains word, 0 otherwise
Simple Query

• Which plays of Shakespeare contain the words **Brutus AND Caesar** but **NOT Calpurnia**?

• One could `grep` all of Shakespeare’s plays for **Brutus** and **Caesar**, then strip out lines containing **Calpurnia**?
  - Slow (for large corpora)
  - **NOT Calpurnia** is non-trivial
  - Other operations (e.g., find the word **Romans** near **countrymen**) not feasible
## Term-document incidence

<table>
<thead>
<tr>
<th></th>
<th>Antony and Cleopatra</th>
<th>Julius Caesar</th>
<th>The Tempest</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Brutus</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>worser</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

*Brutus AND Caesar but NOT Calpurnia*
Answers to query

• Antony and Cleopatra, Act III, Scene ii
  • Agrippa [Aside to DOMITIUS ENOBARBUS]: Why, Enobarbus,
    • When Antony found Julius Caesar dead,
    • He cried almost to roaring; and he wept
    • When at Philippi he found Brutus slain.

• Hamlet, Act III, Scene ii
  • Lord Polonius: I did enact Julius Caesar I was killed i' the
    • Capitol; Brutus killed me.
Bigger corpora

- Consider $n = 1M$ documents, each with about 1K terms.
- Avg 6 bytes/term incl spaces/punctuation
  - 6GB of data in the documents.
- Say there are $m = 500K$ distinct terms among these.
- $500K \times 1M$ matrix has half-a-trillion 0’s and 1’s.
- But it has no more than one billion 1’s.
  - matrix is extremely sparse.
- What’s a better representation?
  - We only record the 1 positions.
Inverted index

- For each term $T$, we must store a list of all documents that contain $T$.
- Do we use an array or a list for this?

<table>
<thead>
<tr>
<th>Brutus</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>128</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calpurnia</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
</tr>
<tr>
<td>Caesar</td>
<td>13</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Inverted index

- Linked lists generally preferred to arrays
  - Dynamic space allocation
  - Insertion of terms into documents easy
  - Space overhead of pointers

Sorted by docID (more later on why).
Indexer steps

- Sequence of (Modified token, Document ID) pairs.

```
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
```

```
<table>
<thead>
<tr>
<th>Term</th>
<th>Doc #</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>
```
• Sort by terms.

Core indexing step.
• Multiple term entries in a single document are merged.
• Frequency information is added.

Why frequency? Will discuss later.
• The result is split into a *Dictionary* file and a *Postings* file.
Query processing

• Consider processing the query:

Brutus AND Caesar

• Locate Brutus in the Dictionary;
  • Retrieve its postings.
• Locate Caesar in the Dictionary;
  • Retrieve its postings.
• “Merge” the two postings:
The merge

- Walk through the two postings simultaneously, in time linear in the total number of postings entries

If the list lengths are $x$ and $y$, the merge takes $O(x+y)$ operations.

**Crucial:** postings sorted by docID.
Boolean queries: Exact match

- Boolean Queries are queries using *AND*, *OR* and *NOT* together with query terms
  - Views each document as a set of words
  - Is precise: document matches condition or not.
- Primary commercial retrieval tool for 3 decades.
- Professional searchers (e.g., lawyers) still like Boolean queries:
  - You know exactly what you’re getting.
Example: WestLaw

- Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992)
- About 7 terabytes of data; 700,000 users
- Majority of users *still* use boolean queries
- Example query:
  - What is the statute of limitations in cases involving the federal tort claims act?
  - LIMIT! /3 STATUTE ACTION /S FEDERAL /2 TORT /3 CLAIM
- Long, precise queries; proximity operators; incrementally developed; not like web search
More general merges

• **Exercise**: Adapt the merge for the queries:
  
  *Brutus* **AND NOT** *Caesar*
  
  *Brutus* **OR NOT** *Caesar*

Q: Can we still run through the merge in time $O(x+y)$?
Query optimization

- What is the best order for query processing?
- Consider a query that is an AND of $t$ terms.
- For each of the $t$ terms, get its postings, then AND together.

**Query**: Brutus AND Calpurnia AND Caesar
Query optimization example

- Process in order of increasing freq:
  - *start with smallest set, then keep cutting further.*

This is why we kept freq in dictionary
Another reason: weighting

Execute the query as *(Caesar AND Brutus) AND Calpurnia.*
More general optimization

- e.g., \((madding \ OR \ crowd) \ AND \ (ignoble \ OR \ strife)\)
- Get freq’s for all terms.
- Estimate the size of each \(OR\) by the sum of its freq’s (conservative).
- Process in increasing order of \(OR\) sizes.
FASTER POSTINGS
MERGES:
SKIP POINTERS
Recall basic merge

• Walk through the two postings simultaneously, in time linear in the total number of postings entries

If the list lengths are \( m \) and \( n \), the merge takes \( O(m+n) \) operations.

Can we do better?
Yes, if index isn’t changing too fast.
Augment postings with skip pointers (at indexing time)

- Why?
  - To skip postings that will not figure in the search results.
- How?
- Where do we place skip pointers?
Query processing with skip pointers

Suppose we’ve stepped through the lists until we process 8 on each list.

When we get to 16 on the top list, we see that its successor is 32.

But the skip successor of 8 on the lower list is 31, so we can skip ahead past the intervening postings.
Where do we place skips?

• Tradeoff:
  • More skips → shorter skip spans ⇒ more likely to skip. But lots of comparisons to skip pointers.
  • Fewer skips → few pointer comparison, but then long skip spans ⇒ few successful skips.
Placing skips

• Simple heuristic: for postings of length $L$, use $\sqrt{L}$ evenly-spaced skip pointers.
• This ignores the distribution of query terms.
• Easy if the index is relatively static; harder if $L$ keeps changing because of updates.

• This definitely used to help; with modern hardware it may not (Bahle et al. 2002)
  • The cost of loading a bigger postings list outweighs the gain from quicker in memory merging
POSITIONAL INDEX - PROXIMITY
Positional indexes

- Store, for each **term**, entries of the form:
  <number of docs containing **term**;
  *doc1*: position1, position2 … ;
  *doc2*: position1, position2 … ;
  etc.>
Positional index example

\(<\textit{be}}: 993427; \\
1: 7, 18, 33, 72, 86, 231; \\
2: 3, 149; \\
4: 17, 191, 291, 430, 434; \\
5: 363, 367, \ldots>\)

- This expands postings storage \textit{substantially}

Which of docs 1, 2, 4, 5 could contain “to be or not to be”?
Processing a phrase query

• Extract inverted index entries for each distinct term: to, be, or, not.

• Merge their doc:position lists to enumerate all positions with “to be or not to be”.
  
  • to:
    
    • 2:1,17,74,222,551; 4:8,16,190,429,433; 7:13,23,191; ...
  
  • be:
    
    • 1:17,19; 4:17,191,291,430,434; 5:14,19,101; ...

• Same general method for proximity searches (within $k$ words)
Proximity queries

- **LIMIT! /3 STATUTE /3 FEDERAL /2 TORT** Here, \( /k \) means “within \( k \) words of”.

- Exercise: How to adapt the linear merge of postings to handle proximity queries?
Positional index size

- Need an entry for each occurrence, not just once per document
- Index size depends on average document size
  - Average web page has <1000 terms
  - SEC filings, books, even some epic poems … easily 100,000 terms
- Consider a term with frequency 0.1%

<table>
<thead>
<tr>
<th>Document size</th>
<th>Postings</th>
<th>Positional postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100,000</td>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>

Why?
Rules of thumb

• A positional index is 2-4 as large as a non-positional index
• Positional index size 35-50% of volume of original text
• Caveat: all of this holds for “English-like” languages
TERM NORMALIZATION
Lemmatization

• Reduce inflectional/variant forms to base form (lemma)
• E.g.,
  • *am, are, is* → *be*
  • *computing* → *compute*
  • *car, cars, car's, cars'* → *car*
• *the boy's cars are different colors* → *the boy car be different color*
• Lemmatization implies doing “proper” reduction to dictionary headword form
  • Need Part-Of-Speech (POS) tagging
Stemming

• Reduce terms to their “roots”/stems before indexing
• “Stemming” suggest crude affix chopping
  • language dependent
  • e.g., automate(s), automatic, automation all reduced to automat.

For example compressed and compression are both accepted as equivalent to compress.

For example compress and compress ar both accepted as equivalent to compress.
Porter’s algorithm

- Commonest algorithm for stemming English
  - Results suggest at least as good as other stemming options
- Conventions + 5 phases of reductions
  - phases applied sequentially
  - each phase consists of a set of commands
  - sample convention: Of the rules in a compound command, select the one that applies to the longest suffix.
Porter algorithm

- **Step 1:** plurals and past participles
  - SSES -> SS  
  - (*v*) ING ->  
  - caresses -> caress  
  - motoring -> motor

- **Step 2:** adj->n, n->v, n->adj, …
  - (m>0) OUSNESS -> OUS  
  - (m>0) ATIONAL -> ATE  
  - callousness -> callous  
  - relational -> relate

- **Step 3:**
  - (m>0) ICATE -> IC  
  - triplicate -> triplic

- **Step 4:**
  - (m>1) AL ->  
  - (m>1) ANCE ->  
  - revival -> reviv  
  - vital -> vital  
  - allowance -> allow

- **Step 5:**
  - (m>1) E ->  
  - (m > 1 and *d and *L) -> single letter  
  - probate -> probat  
  - control -> control
Other stemmers

• Other stemmers exist, e.g., Lovins stemmer
  http://www.comp.lancs.ac.uk/computing/research/stemming/general/lovins.htm
  • Single-pass, longest suffix removal (about 250 rules)
  • Motivated by Linguistics as well as IR

  • Use a dictionary – if a word is in the dict, no change, otherwise, suffix removal

• Full morphological analysis – at most modest benefits for retrieval

• Do stemming and other normalizations help?
  • Often very mixed results: really help recall for some queries but harm precision on others

• Question:
  • Stemming usually remove suffixes. Can we also remove prefixes?
Example (from Croft et al.’s book)

• Original
Document will describe marketing strategies carried out by U.S. companies for their agricultural chemicals, …

• Porter
document describ market strategi carri compani agricultur chemic …

• Krovetz
document describe marketing strategy carry company agriculture chemical …
STOPLIST
Stopwords / Stoplist

• Stopword = word that is not meaning bearer

• Function words do not bear useful information for IR of, in, about, with, I, although, …

• Stoplist: contain stopwords, not to be used as index
  • Prepositions: of, in, from, …
  • Articles: the, a, …
  • Pronouns: I, you, …
  • Some adverbs and adjectives: already, appropriate, many, …
  • Some frequent nouns and verbs: document, ask, …

• The removal of stopwords usually improves IR effectiveness in TREC experiments

• But more conservative stoplist for web search
  • “To be or not to be”

• A few “standard” stoplists are commonly used.
# Stoplist in Smart (571)

<table>
<thead>
<tr>
<th>a</th>
<th>all</th>
<th>and</th>
<th>are</th>
</tr>
</thead>
<tbody>
<tr>
<td>a's</td>
<td>allow</td>
<td>another</td>
<td>aren't</td>
</tr>
<tr>
<td>able</td>
<td>allows</td>
<td>any</td>
<td>around</td>
</tr>
<tr>
<td>about</td>
<td>almost</td>
<td>anybody</td>
<td>as</td>
</tr>
<tr>
<td>above</td>
<td>alone</td>
<td>anyhow</td>
<td>aside</td>
</tr>
<tr>
<td>according</td>
<td>along</td>
<td>anyone</td>
<td>ask</td>
</tr>
<tr>
<td>accordingly</td>
<td>already</td>
<td>anything</td>
<td>asking</td>
</tr>
<tr>
<td>across</td>
<td>also</td>
<td>anyway</td>
<td>associated</td>
</tr>
<tr>
<td>actually</td>
<td>although</td>
<td>anywhere</td>
<td>at</td>
</tr>
<tr>
<td>after</td>
<td>always</td>
<td>anywhere</td>
<td>available</td>
</tr>
<tr>
<td>afterwards</td>
<td>am</td>
<td>apart</td>
<td>away</td>
</tr>
<tr>
<td>again</td>
<td>among</td>
<td>appear</td>
<td>awfully</td>
</tr>
<tr>
<td>against</td>
<td>amongst</td>
<td>appreciate</td>
<td></td>
</tr>
<tr>
<td>ain't</td>
<td>an</td>
<td>appropriate</td>
<td></td>
</tr>
</tbody>
</table>
Discussions

• What are the advantages of filtering out stop words?
  - Discard useless terms that may bring noise
  - Reduce the size of index (recall Zipf law)

• What problems this can create?
  - Difficult to decide on many frequent words: useful in some area but not in some others
  - A too large stoplist may discard useful terms
  - Stopwords in some cases (specific titles) may be useful
  - Silence in retrieval – document cannot be retrieved for this word
TERM WEIGHTING
Assigning Weights

- Weight = importance of the term in a document
  - Also how discriminative it is to distinguish the document from others

- Want to weight terms highly if they are
  - frequent in relevant documents … BUT
  - infrequent in the collection as a whole

- Typical good terms (high weights)
  - Technical terms that only appear in a few documents
  - Specific terms

- Typical weak terms (low weights)
  - General terms
  - Common terms in a language
Assigning Weights

• **tf x idf measure:**
  - term frequency (tf) – to measure local importance
  - inverse document frequency (idf) - discriminativity

\[ T_k = \text{term } k \text{ in document } D_i \]
\[ tf_{ik} = \text{frequency of term } T_k \text{ in document } D_i \]
\[ idf_k = \text{inverse document frequency of term } T_k \text{ in collection } C \]
\[ N = \text{total number of documents in the collection } C \]
\[ n_k = \text{the number of documents in } C \text{ that contain } T_k \]
\[ idf_k = \log(N / n_k) \]
Some common $tf*idf$ schemes

$$tf_{ik} = freq(t_k, D_i)$$
$$tf_{ik} = \log freq(t_k, D_i)$$
$$tf_{ik} = \log (freq(t_k, D_i) + 1)$$
$$tf_{ik} = \log (freq(t_k, D_i) + 1)$$

$$idf_k = \log (N / n_k)$$

$$w_{ik} = tf_{ik} \times idf_k$$
tf x idf (cosine) normalization

- Normalize the term weights (so longer documents are not unfairly given more weight)
  - **normalize** usually means force all values to fall within a certain range, usually between 0 and 1, inclusive.

\[
W_{ik} = \frac{tf_{ik} \log(N / n_k)}{\sqrt{\sum_{k=1}^{t} (tf_{ik})^2 [\log(N / n_k)]^2}}
\]
Questions

• When should the weights of terms be calculated (hint: may require more than one step)?
• Is TF*IDF sufficient? What problems remain?
INDEX COMPRESSION
Corpus size for estimates

- Consider $n = 1$M documents, each with about $L = 1$K terms.
- Avg 6 bytes/term incl spaces/punctuation
  - 6GB of data.
- Say there are $m = 500$K distinct terms among these.
Don’t build the matrix

• 500K x 1M matrix has half-a-trillion 0’s and 1’s.
• But it has no more than one billion 1’s.
  • matrix is extremely sparse.
• So we devised the inverted index
  • Devised query processing for it
• Where do we pay in storage?
• Where do we pay in storage?
Storage analysis (postings)

- First will consider space for postings pointers
- Basic Boolean index only
  - Devise compression schemes
- Then will do the same for dictionary
- No analysis for positional indexes, etc.
Pointers: two conflicting forces

• A term like *Calpurnia* occurs in maybe one doc out of a million - would like to store this pointer using $\log_2 1M \sim 20$ bits.

• A term like *the* occurs in virtually every doc, so 20 bits/pointer is too expensive.
  • Prefer 0/1 vector in this case.
Postings file entry – using gaps

- Store list of docs containing a term in increasing order of doc id.
  - **Brutus**: 33, 47, 154, 159, 202 …
- Consequence: suffices to store gaps.
  - 33, 14, 107, 5, 43 …
- Hope: most gaps encoded with far fewer than 20 bits.
- How about a rare word?
Variable encoding

- For Calpurnia, will use ~20 bits/gap entry.
- For the, will use ~1 bit/gap entry.
- If the average gap for a term is $G$, want to use $\sim \log_2 G$ bits/gap entry.
- **Key challenge**: encode every integer (gap) with $\sim$ as few bits as needed for that integer.
\( \gamma \) codes for gap encoding (Elias)

<table>
<thead>
<tr>
<th>Length</th>
<th>Offset</th>
</tr>
</thead>
</table>

- Represent a gap \( G \) as the pair \(<\text{length}, \text{offset}>\)
- \text{length} is in \( \text{unary} \) and uses \( \lceil \log_2 G \rceil + 1 \) bits to specify the length of the binary encoding of
- \text{offset} = \( G - 2^{\lceil \log_2 G \rceil} \) in binary (remove the first 1)

Recall that the unary encoding of \( x \) is a sequence of \( x \) 1’s followed by a 0.
\(\gamma\) codes for gap encoding

<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></td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>0</td>
<td></td>
<td>0,</td>
</tr>
<tr>
<td>2</td>
<td>110</td>
<td>10</td>
<td>0</td>
<td>10,0</td>
</tr>
<tr>
<td>9</td>
<td>1111111110</td>
<td>1110</td>
<td>001</td>
<td>1110,001</td>
</tr>
</tbody>
</table>

- e.g., 9 represented as \(<1110,001>\)
  - The offset uses 3 bits, the value of offset is 1001
- 2 is represented as \(<10,1>\).
- **Exercise**: does zero have a \(\gamma\) code?
- Encoding \(G\) takes \(2 \lceil \log_2 G \rceil + 1\) bits.
  - \(\gamma\) codes are always of odd length.
Exercise

• Given the following sequence of $\gamma$–coded gaps, reconstruct the postings sequence:

$1110001110101011111101101111011$

From these $\gamma$–decode and reconstruct gaps, then full postings.

$1001 \ 110 \ 11 \ 111011 \ 111$
$1001 \ 1111 \ 10010 \ 1001101 \ 1010100$
What we’ve just done

- Encoded each gap as tightly as possible, to within a factor of 2.
- For better tuning (and a simple analysis) - need a handle on the distribution of gap values.
Zipf’s law

- The $k$th most frequent term has frequency proportional to $1/k$.
- Rank of the term * its frequency $\approx$ constant
- Use this for a crude analysis of the space used by our postings file pointers.
  - Not yet ready for analysis of dictionary space.
Zipf law

Relative frequency

rank
Zipf’s law log-log plot
Rough analysis based on Zipf

• The $i$ th most frequent term has frequency (not count) proportional to $1/i$
• Let this frequency be $c/i$.
• Then $\sum_{i=1}^{500,000} c/i = 1$.
• The $k$ th Harmonic number is $H_k = \sum_{i=1}^{k} 1/i$.
• Thus $c = 1/H_m$, which is $\sim 1/\ln m = 1/\ln(500k) \sim 1/13$.
• So the $i$ th most frequent term has frequency roughly $1/13i$. 
Postings analysis contd.

• Expected number of occurrences of the $i$th most frequent term in a doc of length $L$ is:

$$Lc/i \sim L/13i \sim 76/i \text{ for } L=1000.$$ 

Let $J = Lc \sim 76$.

Then the $J$ most frequent terms are likely to occur in every document. ($J/i \geq 1 \Rightarrow i \leq J$)

Now imagine the term-document incidence matrix with rows sorted in decreasing order of term frequency:
Rows by decreasing frequency

- $J$ most frequent terms.
- $J$ next most frequent terms.
- $J$ next most frequent terms.

- $n$ gaps of ‘1’ each.
- $n/2$ gaps of ‘2’ each (all $i$ such that $\frac{1}{2} \leq J/i < 1$)
- $n/3$ gaps of ‘3’ each. ($1/3 \leq J/i < 1/2$)

etc.
**J-row blocks**

- In the $i$ th of these $J$-row blocks, we have $J$ rows each with $n/i$ gaps of $i$ each.
- Encoding a gap of $i$ takes us $2\log_2 i + 1$ bits ($\gamma$ code)
- So such a row uses space $\sim (2n \log_2 i)/i$ bits ($n/i$ gaps).
- For the entire block, $(2nJ \log_2 i)/i$ bits, which in our case is $\sim 1.5 \times 10^8 (\log_2 i)/i$ bits.

- Sum this over $i$ from $1$ upto $m/J = 500K/76 \sim 6500$. (Since there are $m/J$ blocks.)
Exercise

• Work out the above sum and show it adds up to about 53 x 150 Mbits, which is about 1GByte.
• So we’ve taken 6GB of text and produced from it a 1GB index that can handle Boolean queries!

Make sure you understand all the approximations in our probabilistic calculation.
Caveats

• This is not the entire space for our index:
  • does not account for dictionary storage – next up;
  • as we get further, we’ll store even more stuff in the index.

• Assumes Zipf’s law applies to occurrence of terms in docs.

• All gaps for a term taken to be the same.
More practical caveat

- $\gamma$ codes are neat but in reality, machines have word boundaries – 16, 32 bits etc
  - Compressing and manipulating at individual bit-granularity is overkill in practice
  - Slows down architecture
- In practice, simpler word-aligned compression (see Scholer reference) better
Word-aligned compression

- Simple example: fix a word-width (say 16 bits)
- Dedicate one bit to be a *continuation bit* \(c\).
- If the gap fits within 15 bits, binary-encode it in the 15 available bits and set \(c=0\).
- Else set \(c=1\) and use additional words until you have enough bits for encoding the gap.
## Dictionary and postings files

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

<table>
<thead>
<tr>
<th>Term</th>
<th>N docs</th>
<th>Tot Freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambitious</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>be</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>hath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>i'</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>it</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>let</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>noble</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>so</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>told</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>you</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>with</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

![Diagram](http://example.com/diagram.png)
Inverted index storage (dictionary)

• Have estimated pointer storage

• Next up: Dictionary storage
  • Dictionary in main memory, postings on disk
    • This is common, especially for something like a search engine where high throughput is essential, but can also store most of it on disk with small, in-memory index

• Tradeoffs between compression and query processing speed
  • Time for lookup
  • Time for decompression
How big is the lexicon V?

- Grows (but more slowly) with corpus size
- Empirically okay model (Heaps’ law):
  \[ m = kN^b \]
  - where \( m \)-vocabulary size, \( b \approx 0.5, k \approx 30 – 100; \ N = \# \) tokens
- For instance TREC disks 1 and 2 (2 Gb; 750,000 newswire articles): ~ 500,000 terms
- \( m \) is decreased by case-folding, stemming

**Exercise:** Can one derive this from Zipf’s Law?
  - See http://arxiv.org/abs/1002.3861/
Dictionary storage - first cut

- Array of fixed-width entries
  - 500,000 terms; 28 bytes/term = 14MB.

<table>
<thead>
<tr>
<th>Terms</th>
<th>Freq.</th>
<th>Postings ptr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>999,712</td>
<td></td>
</tr>
<tr>
<td>aardvark</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>....</td>
<td>....</td>
<td></td>
</tr>
<tr>
<td>zzzz</td>
<td>99</td>
<td></td>
</tr>
</tbody>
</table>

Allows for fast binary search into dictionary

20 bytes

4 bytes each
Exercises

• Is binary search really a good idea?
• What are the alternatives?
Fixed-width terms are wasteful

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

<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 = 500K x 8B = 4MB

Pointers resolve 4M positions: $\log_2{4M} = 22\text{bits} = 3\text{bytes}$
Total space for compressed list

- 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
- 500K terms \(\Rightarrow 9.5\)MB

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

- Store pointers to every $k$th on 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>

Save 9 bytes on 3 pointers.

Lose 4 bytes on term lengths.
Net

- Where we used 3 bytes/pointer without blocking
  - $3 \times 4 = 12$ bytes for $k=4$ pointers,
now we use $3+4=7$ bytes for 4 pointers.

Shaved another $\sim 0.5$MB; can save more with larger $k$.

Why not go with larger $k$?
Impact on search

- Binary search down to 4-term block;
- Then linear search through terms in block.
- 8 documents: binary tree ave. = 2.6 compares
- Blocks of 4 (binary tree), ave. = 3 compares

\[
= \frac{1+2 \cdot 2+4 \cdot 3+4}{8} = \frac{1+2 \cdot 2+2 \cdot 3+2 \cdot 4+5}{8}
\]
Total space

• By increasing $k$, we could cut the pointer space in the dictionary, at the expense of search time; space 9.5MB $\rightarrow$ ~8MB

• Net – postings take up most of the space
  • Generally kept on disk
  • Dictionary compressed in memory
Index size

- Stemming/case folding cut
  - number of terms by ~40%
  - number of pointers by 10-20%
  - total space by ~30%
- Stop words
  - Rule of 30: ~30 words account for ~30% of all term occurrences in written text
  - Eliminating 150 commonest terms from indexing will cut almost 25% of space
Extreme compression (see MG)

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

8automata8automate9automatic10automation

→8{automat}a1e2ic3ion

- Encodes *automat*
- Extra length beyond *automat*.

Begins to resemble general string compression.
Extreme compression

Large dictionary: partition into pages

• use B-tree on first terms of pages
• pay a disk seek to grab each page
Effect of compression on Reuters-RCV1

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Size (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>

- Search speed is not taken into account
- See Trotman for a comparison in speed.
Resources for further reading on compression


• Andrew Trotman, Compressing Inverted Files, Information Retrieval, Volume 6, Number 1 / January, 2003, pp. 5-105.

• http://www.springerlink.com/content/n3238225n8322918/fulltext.pdf

• Special issue on index compression, Information retrieval, Volume 3, Number 1 / July, 2000
  http://www.springerlink.com/content/2pbvu3xj98l6/?p=f4b0fc57d855415f9c02aa82ac0061a6&pi=24
Complete procedure of indexing

• For each document
  • For each token
    • If it is stopword, break
    • Stemming $\rightarrow$ term
    • If not in dictionary, add in dictionary: (term, term_id)
    • Increment term frequency, record position
  • Update document frequency
• Compute tf*idf weights
• Create inverted index
• Index compression
References

- Manning et al.: Chap. 1-5
- Croft et al.: Chap. 4-5