Inverted Index

Hongning Wang

CS@UVa
Abstraction of search engine architecture

Indexed corpus
Crawler
Doc Analyzer
Doc Representation
Indexer

Index

Ranking procedure
Feedback
(Query)
Evaluation
User

Query Rep

results
What we have now

- Documents have been
  - Crawled from Web
  - Tokenized/normalized
  - Represented as Bag-of-Words

- Let’s do search!
  - Query: “information retrieval”

<table>
<thead>
<tr>
<th></th>
<th>information</th>
<th>retrieval</th>
<th>retrieved</th>
<th>is</th>
<th>helpful</th>
<th>for</th>
<th>you</th>
<th>everyone</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doc1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Doc2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Complexity analysis

• Space complexity analysis
  – $O(D \times V)$
    • $D$ is total number of documents and $V$ is vocabulary size
  – Zipf’s law: each document only has about 10% of vocabulary observed in it
    • 90% of space is wasted!
  – Space efficiency can be greatly improved by only storing the occurred words

*Solution: linked list for each document*
Complexity analysis

• Time complexity analysis
  – $O(|q| * D * |D|)$

  • $|q|$ is the length of query, $|D|$ is the length of a document

```python
doclist = []
for (wi in q) {
    for (d in D) {
        for (wj in d) {
            if (wi == wj) {
                doclist += [d];
                break;
            }
        }
    }
}
return doclist;
```

Bottleneck, since most of them won’t match!
Solution: inverted index

• Build a look-up table for each word in vocabulary
  – From word to find documents!

*Time complexity:*
• $O(|q| \times |L|)$, $|L|$ is the average length of posting list
• By Zipf’s law, $|L| \ll D$
Structures for inverted index

• Dictionary: modest size
  – Needs fast random access
  – Stay in memory
    • Hash table, B-tree, trie, …

• Postings: huge
  – Sequential access is expected
  – Stay on disk
  – Contain docID, term freq, term position, …
  – Compression is needed

“Key data structure underlying modern IR”
- Christopher D. Manning
Sorting-based inverted index construction

Term Lexicon:
1 the
2 cold
3 days
4 a
...

DocID Lexicon:
1 doc1
2 doc2
3 doc3
...

Parse & Count

<Tuple>: <termID, docID, count>

Sort by docID

Sort by termID

All info about term 1

Merge sort
Sorting-based inverted index

• Challenges
  – Document size exceeds memory limit

• Key steps
  – Local sort: sort by termID
    • For later global merge sort
  – Global merge sort
    • Preserve docID order: for later posting list join

Can index large corpus with a single machine!
Also suitable for MapReduce!
A second look at inverted index

Approximate search: e.g., misspelled queries, wildcard queries

Proximity search: e.g., phrase queries

Dictionary
- information
- retrieval
- retrieved
- is
- helpful
- for
- you
- everyone

Postings
- Doc1
- Doc2

Dynamic index update
Index compression
Dynamic index update

- Periodically rebuild the index
  - Acceptable if change is small over time and penalty of missing new documents is negligible
- Auxiliary index
  - Keep index for new documents in memory
  - Merge to index when size exceeds threshold
    - Increase I/O operation
    - Solution: multiple auxiliary indices on disk, logarithmic merging
Index compression

• Benefits
  – Save storage space
  – Increase cache efficiency
  – Improve disk-memory transfer rate

• Target
  – Postings file
Index compression

• Observation of posting files
  – Instead of storing docID in posting, we store gap between docIDs, since they are ordered
  – Zipf’s law again:
    • The more frequent a word is, the smaller the gaps are
    • The less frequent a word is, the shorter the posting list is
  – Heavily biased distribution gives us great opportunity of compression!

*Information theory*: entropy measures compression difficulty.
Index compression

• Solution
  – Fewer bits to encode small (high frequency) integers
  – Variable-length coding
    • Unary: $x \geq 1$ is coded as $x - 1$ bits of 1 followed by 0, e.g., $3 \Rightarrow 110; 5 \Rightarrow 11110$
    • $\gamma$-code: $x \Rightarrow$ unary code for $1 + \lfloor \log x \rfloor$ followed by uniform code for $x - 2 \lfloor \log x \rfloor$ in $\lfloor \log x \rfloor$ bits, e.g., $3 \Rightarrow 101, 5 \Rightarrow 11001$
    • $\delta$-code: same as $\gamma$-code, but replace the unary prefix with $\gamma$-code. E.g., $3 \Rightarrow 1001, 5 \Rightarrow 10101$
Index compression

• Example

Table 1: Index and dictionary compression for Reuters-RCV1. (Manning et al. Introduction to Information Retrieval)

<table>
<thead>
<tr>
<th>Data structure</th>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text collection</td>
<td>960.0</td>
</tr>
<tr>
<td>dictionary</td>
<td>11.2</td>
</tr>
<tr>
<td>Postings, uncompressed</td>
<td>400.0</td>
</tr>
<tr>
<td>Postings $\gamma$-coded</td>
<td>101.0</td>
</tr>
</tbody>
</table>

Compression rate: $(101+11.2)/960 = 11.7\%$
Search within in inverted index

• Query processing
  – Parse query syntax
    • E.g., Barack AND Obama, orange OR apple
  – Perform the same processing procedures as on documents to the input query
    • Tokenization->normalization->stemming->stopwords removal
Search within in inverted index

• Procedures
  – Lookup query term in the dictionary
  – Retrieve the posting lists

• Operation
  • AND: intersect the posting lists
  • OR: union the posting list
  • NOT: diff the posting list
Search within in inverted index

• Example: AND operation

Time complexity: $O(|L_1| + |L_2|)$

*Trick for speed-up:* when performing multi-way join, starts from lowest frequency term to highest frequency ones.
Phrase query

• “computer science”
  – “He uses his computer to study science problems” is not a match!
  – We need the phase to be exactly matched in documents
  – N-grams generally does not work for this
    • Large dictionary size, how to break long phrase into N-grams?
  – We need term positions in documents
    • We can store them in inverted index
Phrase query

• Generalized postings matching
  – Equality condition check with requirement of position pattern between two query terms
    • e.g., T2.pos-T1.pos = 1 (T1 must be immediately before T2 in any matched document)
  – Proximity query: |T2.pos-T1.pos| \leq k
More and more things are put into index

• Document structure
  – Title, abstract, body, bullets, anchor

• Entity annotation
  – Being part of a person’s name, location’s name
Spelling correction

• Tolerate the misspelled queries
  – “barck obama” -> “barack obama”

• Principles
  – Of various alternative correct spellings of a misspelled query, choose the nearest one
  – Of various alternative correct spellings of a misspelled query, choose the most common one
Spelling correction

• Proximity between query terms
  – Edit distance
    • Minimum number of edit operations required to transform one string to another
    • Insert, delete, replace
  • Tricks for speed-up
    – Fix prefix length (error does not happen on the first letter)
    – Build character-level inverted index, e.g., for length 3 characters
    – Consider the layout of a keyboard
      » E.g., ‘u’ is more likely to be typed as ‘y’ instead of ‘z’
Spelling correction

• Proximity between query terms
  – Query context
    • “flew form Heathrow” -> “flew from Heathrow”
  – Solution
    • Enumerate alternatives for all the query terms
    • Heuristics must be applied to reduce the search space
Spelling correction

• Proximity between query terms
  – Phonetic similarity
    • “herman” -> “Hermann”
  – Solution
    • Phonetic hashing – similar-sounding terms hash to the same value
What you should know

• Inverted index for modern information retrieval
  – Sorting-based index construction
  – Index compression

• Search in inverted index
  – Phrase query
  – Query spelling correction