CS 589 Fall 2020

Information retrieval infrastructure

Instructor: Susan Liu
TA: Huihui Liu

Stevens Institute of Technology
Information Retrieval Infrastructure
Inverted index

• In Lecture 2, we learned retrieval models
  • Compute score(q, d)
  • Select the d that maximizes score(q, d)

• In an industry scale search engine, there could be trillions of q’s and billions of d’s
  • For each query, search time complexity = $O(|D|)$
  • Solution for faster retrieval: inverted index
Inverted index

1: Winter is coming.
2: Ours is the fury.
3: The choice is yours.

<table>
<thead>
<tr>
<th>term</th>
<th>freq</th>
<th>documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>choice</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>coming</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fury</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>is</td>
<td>3</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>ours</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
<td>2, 3</td>
</tr>
<tr>
<td>winter</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>yours</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Dictionary

Postings

time complexity: $O(\#\text{unique words in q} \times \text{avg}_\text{len(postings lists)})$

$\ll |D|$
Problems with inverted indexing

- Data processing
  - Choosing the unit for indexing
  - Determining the vocabulary

- Constructing/speeding up inverted index
  - Skipping index
  - Prefix indexing
  - Indexing with blocks
  - MapReduce

- Index compression

- Other issues
  - Indexing position
  - Spelling correction
Choosing the correct unit for indexing

- Documents often consist of sub documents
  - e.g., email contains multiple attached documents

- Trade-off on the unit size
  - Smaller units: missing important passages
  - Larger unit: gets spurious matches, e.g., text messages, gold mining...
Determining the vocabulary

- **Tokenization**
  
  Input: Friends, Romans, Countrymen, lend me your ears;
  Output: **Friends Romans Countrymen lend me your ears**

- **o’neil, aren’t, C#**

- **Dropping stop words**
  - **Stop words are common terms**
  - **Web search engines generally do not use stop words!**

<table>
<thead>
<tr>
<th>A</th>
<th>It</th>
<th>These</th>
</tr>
</thead>
<tbody>
<tr>
<td>About</td>
<td>Its</td>
<td>They</td>
</tr>
<tr>
<td>Again</td>
<td>Itsel</td>
<td>This</td>
</tr>
<tr>
<td>All</td>
<td>Just</td>
<td>Those</td>
</tr>
<tr>
<td>Almost</td>
<td>km</td>
<td>Thus</td>
</tr>
<tr>
<td>Also</td>
<td>Made</td>
<td>To</td>
</tr>
<tr>
<td>Although</td>
<td>Mainly</td>
<td>Upon</td>
</tr>
<tr>
<td>Always</td>
<td>Make</td>
<td>Use</td>
</tr>
<tr>
<td>An</td>
<td>May</td>
<td>Used</td>
</tr>
<tr>
<td>And</td>
<td>mg</td>
<td>Using</td>
</tr>
<tr>
<td>Another</td>
<td>Might</td>
<td>Various</td>
</tr>
<tr>
<td>Any</td>
<td>ml</td>
<td>Very</td>
</tr>
<tr>
<td>Are</td>
<td>mm</td>
<td>Was</td>
</tr>
<tr>
<td>As</td>
<td>Most</td>
<td>We</td>
</tr>
<tr>
<td>At</td>
<td>Mostly</td>
<td>Were</td>
</tr>
</tbody>
</table>
Determining the vocabulary

• Normalization
  • Abbrev: USA vs. United states of America
  • Case:
    • Cat -> cat
    • SAT -> sat

• Stemming/lemmatization
  • singing -> sing, cars -> car, sat -> sit
  • porter stemmer, snowball stemmer
Speeding up: skipping lists

- Finding the intersection of two post listings
  - Without skip: $O(m + n)$

```
i, j = 0, 0
while i < m and j < n:
    if arr1[i] < arr2[j]:
        i += 1
    elif arr2[j] < arr1[i]:
        j += 1
    else:
        print(arr2[j])
        j += 1
        i += 1
```
Speeding up: prefix indexing

- Speeding up the indexing using prefix tree
  - time complexity: $O(#\text{unique words in q} \times \text{avg}_\text{len(postings lists)})$
Constructing inverted index: hardware basics

• Decisions on an IR system largely depends on the hardware which the system runs on

• **Chunks:**
  • Splitting data into more chunks takes more *seek time*

• **Blocks**
  • *Accessing data in memory >> accessing data on disk*
  • *Constructing* inverted index using blocks
  • Typical IR system: GBs of memory, disk space orders of magnitude larger
Block sort-based indexing

- Indexing large corpus
  - Reuters-RCV1: 2.5GB = 2.5 x 10^9Bytes, 1 billion
  - Today’s text corpus contains petabytes of data: 10^15Bytes
  - Memory << size of corpus

- Index each block using memory
- Write each blocks’ index into disk
- Merge all inverted indices
Block sort-based indexing

1. segment corpus into blocks
2. sort (term, doc) pair in mem
3. store the pairs in disk
4. merge all pairs into final index

BSBINDEXCONSTRUCTION()
1. \( n \leftarrow 0 \)
2. while (all documents have not been processed)
3. \( \text{do } n \leftarrow n + 1 \)
4. \( \text{block } \leftarrow \text{PARSENEXTBLOCK()} \)
5. \( \text{BSBI-INVERT(block)} \)
6. \( \text{WRITEBLOCKTO_DISK(block, f_n)} \)
7. \( \text{MERGEBLOCKS}(f_1, \ldots, f_n; f_{\text{merged}}) \)
Single-pass in-memory indexing

- Segment corpus into blocks
- Write block into posting lists & dict
- Store the postings lists in disk
- Merge all blocks into final index

- Handling posting lists directly
- Eliminating the expensive sorting in BSBI
- Leveraging compression
Handling web scale indexing

• Web-scale indexing must use clusters of servers
  • Google had 1 million servers in 2011

• Fault tolerance of a massive data center
  • If a non-fault tolerance system has 1000 nodes, each has 99.9% uptime, then 63% of the time one or more servers is down

• Solution
  • Maintain a “master” server
  • Break indexing into parallel tasks
  • Assign each task to an idle machine
Map-reduce

- Master assigns split to idle machine
- Parser emits (term, doc) pair
- Inverter merges partitions in inverter
- Complete the index
Examples of map-reduce

map: $d_2 : C$ died. $d_1 : C$ came, C c’ed.

\[
\langle \langle C, d_2 \rangle, \langle \text{died}, d_2 \rangle, \langle C, d_1 \rangle, \langle \text{came}, d_1 \rangle, \langle C, d_1 \rangle, \langle \text{c’ed}, d_1 \rangle \rangle
\]

reduce: $\langle \langle C, (d_2, d_1, d_1) \rangle, \langle \text{died}, (d_2) \rangle, \langle \text{came}, (d_1) \rangle, \langle \text{c’ed}, (d_1) \rangle \rangle$

\[
\langle \langle C, (d_1:2, d_2:1) \rangle, \langle \text{died}, (d_2:1) \rangle, \langle \text{came}, (d_1:1) \rangle, \langle \text{c’ed}, (d_1:1) \rangle \rangle
\]
MapReduce: Industry practice

• Term partition vs. document partition
  • Term-partitioned: one machine handles a subrange of terms
  • Document-partitioned: one machine handles a subrange of documents

• Most industry search engine use document-partitioned index
  • Better load balancing (why?)

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat
jeff@google.com, sanjay@google.com

Google, Inc.
Dynamic indexing

• Document collection are updated all the time

• How to handle dynamic indexing?
  • Maintain one-big index
  • New document goes to auxiliary “smaller” index
  • Search across both, merge results
  • Periodically merge the two indices
  • Deletion: maintain bitvectors of deleted documents
Real time search of Twitter

- Requires high real time search
  - Low latency, high throughput query evaluation
  - High ingestion rate and immediate data availability
  - Concurrent reads and writes of the index

- Solution: using segments
  - Each segment consists of $2^{32}$ tweets (in memory)
  - New posts are appended to the posting lists
  - Only one segment can be written to at each time
Index compression

• Why compression?
  • Using less disk space
  • Compressing dictionary
    • Allowing the dictionary to be stored in memory
  • Compressing posting files
    • Reducing disk space

• Zipf’s law
  • The ith most frequent term has frequency proportional to $1/i$
Dictionary compression

- Most of the space in the table is wasted
  - Most words are no 20 bytes
  - Table storage = 28N

<table>
<thead>
<tr>
<th>Terms</th>
<th>Freq.</th>
<th>Postings ptr.</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>

20 bytes | 4 bytes each
Dictionary-as-a-string

- Table storage = 11N

- How to further improve the storage space?
  - Instead of storing absolute term pointers, store the gaps
Dictionary-as-a-string

- Table storage = 8N + 3N * (7/12) = 9.75N < 11N
- Trade-off between skipping more vs. skipping less

<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.
Postings compression

• Observations of posting files
  • Instead of storing docID, store gaps
  • Brutus: 2,4,8,3,4,5,15
  • Binary seq: 10,100,1000,11,100,101,1

• Prefix encoding
  • Binary encoding such that the sequence can be uniquely decoded
  • e.g., Huffman encoding
  • Unary encoding: \{2:110,4:11110, \ldots\}
  • A uniquely decodable seq: 110111101111111101110\ldots
Postings compression

- Problem with unary encoding
  - Too long!

- Gamma code of 13: 1110,101
  - Length: 1110
  - Offset: 13 → 1101 → 101

- We can prove gamma code is uniquely decodable

- Gamma code compression rate: 11.7%
Indexing Position

- Indexing the position of word within the document
- Intersection algorithm finds where the two terms appear between within $k$ words

Brutus: 2:<0>, 4: <429, 433>, 8: <150>, …

Ceasar: 1:<10>, 2: <29>, 8: <17, 250>, …
Spelling correction

• Edit distance
• k-gram index for spelling correction
• context sensitive spelling correction
**Edit distance**

- Dynamic programming: $O(|s_1| \times |s_2|)$

```
EDITDISTANCE(s1, s2)
1  init m[i, j] = 0
2  for i ← 1 to |s1|
3    do m[i, 0] = i
4  for j ← 1 to |s2|
5    do m[0, j] = j
6  for i ← 1 to |s1|
7    do for j ← 1 to |s2|
8      do m[i, j] = min(m[i - 1, j - 1] + if (s1[i] = s2[j]) then 0 else 2
9         m[i - 1, j] + 1,
10        m[i, j - 1] + 1)
11  return m[|s1|, |s2|]
```

Levenshtein distance
k-gram indexes for spelling correction

• Running DP on all pairs of words is time consuming

• Leveraging k-gram index to speed up spelling correction
  • boardroom vs. bord

boarder: 3
boardroom: 2
aboard: 2
ardent: 1

...
Context sensitive spelling correction

• How to correct “flew form healthrow”?  
  • All three words are spelled correctly  
  • Enumerating each character: the space is large  
  • Solution: using logs of queries, e.g., flew from vs. fled fore

Li et al. A generalized hidden Markov model with discriminative training for query spelling correction. SIGIR 2012
PageRank

• How to rank webpages?
  • Using retrieval models: only captures relevance

• Capturing quality of web pages:
  • Based on how often the page is cited
  • Intuition: a popular website (e.g., Google) would be cited by a lot of other webpages
PageRank


\[ PR(p_i) = \frac{1 - d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)} \]

- Favors pages that are highly cited, and pages cited by highly cited pages

1/2 probability of randomly walking into B
PageRank

• Assign each node an initial page rank

• Repeat until convergence
  • Calculate the page rank of each node using the equation

\[
PR(p_i) = \frac{1 - d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}
\]
Problems of page rank

- rich gets richer
- Google bombing
HITS

- Hubs: compilations of a broad catalog of information that led users direct to other authoritative pages

- Authorities: a page that is linked by many different hubs
HITS

- Repeat k times
  - Update hub score: $v = A^T u$
  - Update authority score: $u = A^T v$
Search engine tools

• Apache Lucene
  • Free and open search engine library
  • First developed in 1999

• ElasticSearch
  • A search engine
  • based on Lucene
ElasticSearch

- Using a REST api
Homework 2: Using ElasticSearch to build a search engine

• Build an inverted index

• Evaluate three search algorithm’s performance
  • TF-IDF
  • BM25
  • Dirichlet-LM