Text Technologies for Data Science
INFR11145

Indexing (2)

Instructor: Walid Magdy

Lecture Objectives

• Learn more about indexing:
  • Structured documents
  • Extent index
  • Index compression
• Data structure
• Wild-char search and applications

* You are not asked to implement any of the content in this lecture, but you might think of using some for your course project 😊
Structured Documents

- Document are not always flat:
  - Meta-data: title, author, time-stamp
  - Structure: headline, section, body
  - Tags: link, hashtag, mention

- How to deal with it?
  - Neglect!
  - Create separate index for each field
  - Use “extent index”

Extent Index

- Special “term” for each element/field/tag
  - Index all terms in a structured document as plain text
  - Terms in a given field/tag get special additional entry
  - Posting: spans of window related to a given field
  - Allows multiple overlapping spans of different types

<table>
<thead>
<tr>
<th>Term</th>
<th>Spans</th>
</tr>
</thead>
<tbody>
<tr>
<td>he</td>
<td>1,1   1,5 2,1 3,3 4,3 5,1</td>
</tr>
<tr>
<td>drink</td>
<td>1,8   2,4 2,6 3,8 4,3 4,5 5,6</td>
</tr>
<tr>
<td>ink</td>
<td>3,8  4,2 5,8</td>
</tr>
<tr>
<td>pink</td>
<td>4,8  5,7</td>
</tr>
<tr>
<td>Link</td>
<td>3,1:2 4,1:4 5,7:8</td>
</tr>
</tbody>
</table>

D1: He likes to wink, he likes to drink
D2: He likes to drink, and drink, and drink
D3: The thing he likes to drink is ink
D4: The ink he likes to drink is pink
D5: He likes to wink, and drink pink ink
Using Extent

• Doc: 1 →
Headline: “Information retrieval lecture”
Text: “this is lecture 6 of the TTDS course on IR”

• Query → Headline: lecture

Index Compression

• Inverted indices are big
  • Large disk space → large I/O operations
• Index compression
  • Reduce space → less I/O
  • Allow more chunks of index to be cached in memory
• Large size goes to:
  • terms? document numbers?
  • Ideas:
    • Compress document numbers, how?
**Delta Encoding**

- Large collections \(\rightarrow\) large sequence of doc IDs
  - e.g. Doc IDs: 1, 2, 3, ... 66,032, ..., 5,323,424,235
- Large ID number \(\rightarrow\) more bytes to store
  - 1 byte: 0\(\rightarrow\)255
  - 2 bytes: 0 \(\rightarrow\) 65,535
  - 4 bytes: 0 \(\rightarrow\) 4.3 B
- Idea: delta in ID instead of full ID
  - Very useful, especially for frequent terms

\[
\begin{array}{c}
\text{term} \longrightarrow \ldots \quad 100002 \quad 100007 \quad 100008 \quad 100011 \quad 100019 \quad \ldots \\
\text{term} \longrightarrow \ldots \quad ? \quad 5 \quad 1 \quad 3 \quad 7 \quad \ldots \quad 321 \quad 15 \quad 2 \quad \ldots 
\end{array}
\]

**v-byte Encoding**

- Have different byte storage for each delta in index
  - Use fewer bits to encode
  - High bit in a byte \(\rightarrow\) 1/0 = terminate/continue
  - Remaining 7 bits \(\rightarrow\) binary number
- Examples:
  - "6" \(\rightarrow\) 10000110
  - "127" \(\rightarrow\) 11111111
  - "128" \(\rightarrow\) 0000001100000000
- Real example sequence:
  \[
  100010100000001100001010000111
  \]
Index Compression

• There are more sophisticated compression algorithms:
  • Elias gamma code

• The more compression
  • Less storage
  • More processing

• In general
  • Less I/O + more processing > more I/O + no processing
    “>” = faster
  • With new data structures, problem is less severe

Dictionary Data Structures

• The dictionary data structure stores the term vocabulary, document frequency, pointers to each postings list ...

• For small collections, load full dictionary in memory. In real-life, cannot load all index to memory!
  • Then what to load?
  • How to reach quickly?
  • What data structure to use for inverted index?
Hashes

- Each vocabulary term is hashed to an integer
- Pros
  - Lookup is faster than for a tree: O(1)
- Cons
  - No easy way to find minor variants:
    - judgment/judgement
  - No prefix search
  - If vocabulary keeps growing, need to occasionally do the expensive operation of rehashing everything

Trees: Binary Search Tree

```
   a-m
  /   \
 a-hu  hy-m
  |     |
  |     |
  n-z  n-sh
  /   /   \
 s-d  s-h  s-i
```

...
Every internal node has a number of children in the interval \([a,b]\) where \(a, b\) are appropriate natural numbers, e.g., \([2,4]\).

**Pros?**
- Solves the prefix problem (terms starting with "ab")

**Cons?**
- Slower: \(O(\log M)\) [and this requires balanced tree]
- Rebalancing binary trees is expensive
  - But B-trees mitigate the rebalancing problem
Wild-Card Queries: *

- mon*: find all docs containing any word beginning “mon”.
- Easy with binary tree (or B-tree) lexicon
- *mon: find words ending in “mon”: harder
  - Maintain an additional B-tree for terms backwards.
- How can we enumerate all terms meeting the wild-card query pro*cent ?
- Query processing: se*ate AND fil*er ?
  - Expensive

Permuterm Indexes

- Transform wild-card queries so that the * occurs at the end
- For term hello, index under:
  - hello$, ello$h, llo$he, lo$hel, o$hell, $hello
  where $ is a special symbol.
- Rotate query wild-card to the end
- Queries:
  - X lookup on X$
  - X* lookup on $X*
  - *X lookup on
  - X*Y lookup on
- Index Size?
Character n-gram Indexes

- Enumerate all n-grams (sequence of \(n\) chars) occurring in any term
  - e.g., from text “April is the cruelest month” we get the 2-grams (bigrams) →
    
    \[$a, ap, pr, ri, il, l$, $i, is, s$, $th, he, e$, $c, cr, ru, ue, el, le, es, st, t$, $m, mo, on, nt, h$\]
  - $\$$ is a special word boundary symbol
  - Maintain a second inverted index from bigrams to dictionary terms that match each bigram.
    - Character n-grams → terms
    - terms → documents

Character n-gram Indexes

- The \(n\)-gram index finds terms based on a query consisting of \(n\)-grams (here \(n=2\)).

\[
\begin{align*}
    \text{Wild card query} & \quad \text{Find possible terms} \quad \text{Filter unmatching terms} \quad \text{Search collection for all terms} \quad \text{Documents} \\
    \text{Index of char bigrams} & \quad \text{Collection index of terms}
\end{align*}
\]

\[
\begin{align*}
    $m & \quad \text{mace, madden} \\
    mo & \quad \text{among, amortize} \\
    on & \quad \text{almond, among}
\end{align*}
\]
Character n-gram Indexes: Query time

- **Step 1**: Query mon* → $m$ AND mo AND on
  - It would still match moon.
- **Step 2**: Must post-filter these terms against query.
  - Phrase match, or post-step1 match
- **Step 3**: Surviving enumerated terms are then looked up in the term-document inverted index.
  → Montreal OR monster OR monkey
- Wild-cards can result in expensive query execution (very large disjunctions…)

Character n-gram Indexes: Applications

- Spelling Correction
  - Create n-gram representation for words
  - Build index for words:
    - Dictionary of words → documents (each word is a document)
    - Character n-grams → terms
  - When getting a search term that is misspelled (OOV or not frequent), find possible corrections
    - Possible corrections = most matching results

Query: elegant → $e$e $el$el $le$le $ep$ep $pg$pg $ga$ga $an$an $nt$nt $t$$t$
Results:
  - elegant → $e$e $el$el $le$le $eg$eg $ga$ga $an$an $nt$nt $t$$t$
  - elephant → $e$e $el$el $ep$ep $ph$ph $ha$ha $an$an $nt$nt $t$$t$
Character n-gram Indexes: Applications

- Char n-grams can be used as direct index terms for some applications:
  - Arabic IR, when no stemmer/segmenter is available
  - Documents with spelling mistakes: OCR documents

- Word char representation can be with multiple n’s
  - “elephant” → 2/3-gram →
    “$e\ e\ l\ e\ p\ h\ a\ n\ nt\ t$ $e\ l\ e\ p\ e\ p\ h\ a\ n\ ant\ nt$”

The children behaved well
Her children are cute

Document: Elepbant → $e\ e\ l\ e\ p\ b\ a\ n\ nt\ t$
Query: Elephant → $e\ e\ l\ e\ p\ h\ a\ n\ nt\ t$

Summary

- Index can be multilayer
  - Extent index (multi-terms in one position in document)

- Index does not have to be formed of words
  - Character n-grams representation of words

- Two indexes are sometimes used
  - Index of character n-grams to find matching words
  - Index of terms to search for matched words
Resources

- Text book 1: Intro to IR, Chapter 3.1 – 3.4
- Text book 2: IR in Practice, Chapter 5