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>1,1</th>
<th>1,5</th>
<th>2,1</th>
<th>3,3</th>
<th>4,3</th>
<th>5,1</th>
</tr>
</thead>
<tbody>
<tr>
<td>he</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>drink</td>
<td>1,8</td>
<td>2,4</td>
<td>2,6</td>
<td>2,8</td>
<td>3,6</td>
<td>4,5</td>
</tr>
<tr>
<td>ink</td>
<td>3,8</td>
<td>4,2</td>
<td>5,8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pink</td>
<td></td>
<td></td>
<td>4,8</td>
<td>5,7</td>
<td></td>
<td></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, \ldots 66,032, \ldots, 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

<table>
<thead>
<tr>
<th>Term</th>
<th>\ldots</th>
<th>100002</th>
<th>100007</th>
<th>100008</th>
<th>100011</th>
<th>100019</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term</td>
<td>\ldots</td>
<td>?</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

**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" \(\rightarrow\) 00000001\(\rightarrow\)00000000
- Real example sequence:

```
10000101 00000000 11000001 10000111
0000101 00000001 0000010 000111
```

\[ 5 \rightarrow 130 \rightarrow 7 \]
**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**

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

**Trees**

- **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) → $a, ap, pr, ri, il, l$, $i, is, s$, $t, th, he, e$, $c, cr, ru, ue, el, le, es, st, t$, $m, mo, on, nt, h$
  - $\$ 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).
  - Wild card query
  - Find possible terms
  - Filter unmatching terms
  - Search collection for all terms
  - Documents

Index of char bigrams

Collection index of terms
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 → $e el le ep pg ga an nt t$
  Results:
  - elegant → $e el le eg ga an nt t$
  - elephant → $e el le ep ph ha an nt t$
**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 by with multiple n’s
  - “elephant” → 2/3-gram →
    “$e el le ep ph ha an nt t$ $e el $ele lep eph pha han ant nt$”

The children behaved well
Her children are cute

Document: Elepbant → $e el le ep pb ba an nt t$
Query: Elephant → $e el le ep ph ha an nt t$

**Summary**

- Index can by 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