**Text Technologies**

**Indexing**

Victor Lavenko

Indexing

- What makes search engines fast
  - Infeasible to “scan” or even do fast string-matching
- Analogy: index at the back of a book
  - For each keyword = list of “relevant” pages
  - Locate matches in sub-linear time
- IR index: all words (+phrases), all pages
  - Specialized data structure, different from RDB
  - Many specific optimizations

Document vectors

- Represent documents as vectors
  - One entry per word (10^8 dims, most=0)
  - Dimensions = numeric word ids
  - Value: #times word observed in a doc
  - Collection = matrix

<table>
<thead>
<tr>
<th>Term</th>
<th>Drink</th>
<th>Ink</th>
<th>Pink</th>
<th>Thing</th>
<th>Wink</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drink</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Ink</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Pink</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Thing</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Wink</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Linear merge

- Find documents matching query (ink, wink)
  - Load inverted lists for all query words
  - Linear merge: O(n)
  - N ... total number of items in the two lists
  - f) scoring function: how well this doc matches the query
  - ink = 31
d
  - wink = 11

Scoring Functions

- Specify which docs are matched
  - In: counts of query words in a doc
  - Out: ranking score
  - How well doc matches query
  - 0 if document does not match
- Examples of scoring functions
  - Boolean AND: $f(Q,D) = \prod_{w_i \in Q} n_{w_i \in D}$
  - 1 if all query words are present
  - If JDF weighted sum: $f(Q,D) = \sum_{w_i \in Q} \frac{n_{w_i \in D}}{n_{w_i \in D^+}} \log \frac{IC_{w_i}}{DF_{w_i}}$

Sparsification

- Inverted lists very sparse
  - thing $\rightarrow$ 0 0 1 0 0
- Zipf half the terms will occur only once
- Dimensionality = # docs in collection (huge)
- Sparse representation:
  - (id, value) tuples
    - Sorted by id
    - Compact
  - Very fast
  - Linear merge

Overview

- Indexing
  - Forward and inverted indices
  - Sparse representation
  - Scoring functions
  - Proximity indices
  - Field / extent indices
- Index compression
  - Query execution
  - Index construction
Phrases and proximity

- Want to find "pink ink" as a phrase
- Using regular index:
  - match #and(pink,ink) →
  - scan matching documents for string (slow)
- Index all bi-grams as words
  - can approximate 'drink pink ink'
  - fast, but index size explodes
  - inflexible: can't do #5(pink,ink)
- Construct proximity index

Proximity index

- Embed position information in inverted lists
  - called positional/proximity index (prox-list)
  - handles arbitrary phrases, windows
  - key to "rich" indexing: structure, fields, tags, ...

Using proximity

- Query: want to find adjacent "pink ink"
- Same approach: linear merge
  - compare docs under pointers
  - if match → check: pos(ink) - pos(pink) = 1
  - "near" operator:

Structure and tags

- Documents are not always flat
  - meta-data: title, author, date
  - structure: part, chapter, section, paragraph
  - tags: named entity, link, translation
- Options for dealing with structure:
  - create separate index for each field (a-la SQL)
  - push structure into index values
  - construct extent index

Extent index

- Special "term" for each element / field / tag
  - spans a region of text, words in span belong to field
  - allows multiple overlapping spans of diff. types
  - similar to stand-off annotation formats (Attis)

Using extents

- Query: find an ink-related hyper-link
- Same approach as with proximity
  - only now "tag" and "word" must have distance = 0
  - linear merge, match when position falls into extent
  - amenable to all optimizations

Overview

- Indexing
- Index compression
  - delta encoding
  - v-byte encoding
  - skip pointers
- Query execution
- Index construction

Index compression

- Inverted indices are big
  - take lots of disk space, I/O relatively slow
- Index compression (SE section 5.4)
  - reduce space, improve I/O time, CPU is fast
  - specialized methods → simple and very fast
  - basic ideas:
    - convert large numbers into small ones
    - represent numbers with as few bits as possible

Delta encoding

- Inverted list: sequence of docids
  - docids are large numbers (> 10^9)
  - differences between subsequent numbers can be small
  - replace numbers with deltas (will be compressed)
- Excellent for frequent terms
  - infrequent terms → small proportion of the index
Fast execution: query caching

- **Idea:** hash("query") \(\rightarrow\) \{top k results\}
  - most common queries answered instantly
  - space-efficient: 50% users examine top 20 results
  - storing more: allows re-use in longer queries

- **Concerns:**
  - Zipf's law: 50% queries are unique
  - competes with index cache (if in memory)
    - caching regular indices more productive
  - index updates invalidate cache

Fast execution: heuristics (1)

- **Skip pointers:**
  - doc-at-a-time only, conjunctive or proximity queries
  - iterate over short list \(\{m \text{ items}\}\), search long \(\{n \text{ items}\}\)
  - long list: binary search + skip + decompress until match
  - \(O(\frac{m}{n}c^2)\) or \(O(m(\frac{n}{c} + c^2))\) if compressed

- **Early termination:** user looks at \(< k\) results
  - estimate user's acceptance "threshold" \(t\)
  - easy for Boolean, proximity queries
  - doc-at-a-time: stop after \(k\) matches with score \(> t\)

Fast execution: heuristics (2)

- **Top-docs**
  - keep \(K\) "best" docs in each list (by score)
  - use for ranking, fall back to full index if needed
  - \(K\) may vary across terms / queries / users

- **Prioritize processing** (term-at-a-time)
  - arbitrary ordering of terms in query, docs in index
  - keep indices sorted by decreasing partial score
  - process best docs first, stop early (top-docs w. flexible \(K\))
  - process most significant terms first (usually rare)
  - stop when subsequent terms unlikely to affect ranking

Fast execution: Boolean AND

- **One very long, one short list (SE 5.4.6)**
  - \(O(m+n)\) even though \(m < n\)
  - loop over short list, binary search long:
    - \(O(m \log n)\) ... use only if \(m < n / \log n\)
  - works only if scoring function \(=\) AND (all query words)
  - breaks if list compressed: use skip pointers

\[
\begin{align*}
\text{big} & \rightarrow \{5, 11, 13, 20, 21, 37, 51, 52, 53, 59, 60\} \\
\text{aardvark} & \rightarrow \{51\}
\end{align*}
\]

Index construction: naive approach

- Document vectors \(\rightarrow\) inverted lists
  - assume partial index (documents D1-D4)
  - new document: D5: He likes to wink and drink pink ink.

\[
\begin{align*}
\text{he} & \rightarrow \{1/2, 2/1, 3/1, 4/1, 5/1\} \\
\text{ink} & \rightarrow \{3/1, 4/1, 6/1\} \\
\text{pink} & \rightarrow \{4/1, 5/1\} \\
\text{thing} & \rightarrow \{3/1\} \\
\text{wink} & \rightarrow \{1/1, 5/1\} \\
\text{likes} & \rightarrow \{5/1\}
\end{align*}
\]

Index merging (1)

- Each batch of documents \(\rightarrow\) partial index
  - fits in memory, sorted by term, then document id

\[
\begin{align*}
\text{D4: The ink he likes to drink is pink} \\
\text{D5: He likes to wink and drink pink ink.}
\end{align*}
\]

Query execution: structured

- Complex request: "wink" and "ink" in a title
- Smart iterator (next) at each node
  - calls next() of "lower" child until constraint satisfied
- returns: position in index + extent (if a match)
- leaves: indices

- Indexing
- Index compression
- Query execution
- Index construction
  - naive / in-memory algorithm
  - index merging
  - indexing with MapReduce
  - distributed search

Skip pointers

- Problem with compression
  - can't randomly access list without decompressing
- Skip pointers:
  - "checkpoints" in the compressed list
  - "jump" to pre-determined points, process from there

\[
\begin{align*}
\text{ink} & \rightarrow \ldots 1002 1007 1008 1011 1019 1020 \ldots \\
\text{ink} & \rightarrow 1 1007 2007 \ldots 2 5 1 3 8 1 \ldots
\end{align*}
\]

Compressed inverted list
Indexing (2)
- Merge partial indices into one index
  - in-memory priority queue
  - disk-intensive
  - partial indices independent → parallelize

Distributed indexing
- MapReduce architecture (Google)
  - task split into many small parts
  - turned out to thousands of cheap PCs
  - map: PC solves its part, emits (key: value) pairs
  - shuffle: pairs grouped and routed to reducers
  - reduce: partial results combined

- Open source: Hadoop, Galago

Indexing with MapReduce
- Split into M sets of docs (16-128MB per set)
- Mapper:
  - parse doc.set, produce list of key-value pairs
  - { key = "term", value = ("doc", count/position) }
- Shuffle into R pools: HASH[key] modulo R
- Reducer:
  - reads pairs (grouped by key), sort / merge
  - inverted lists stored in files

Distributed search
- MapReduce: compute in parallel, aggregate
- Idea: keep parts of the index on different PCs
  - broadcast query to all/some index servers
  - each server constructs, sends back ranked list
  - merge ranked list, return result

Summary (1)
- Indexing: collection is a matrix
  - rows = document vectors, columns = inverted lists
  - Zipf’s law → sparse representation for everything
    - linear merge is a fundamental operation
    - phrases / structure → proximity / extent indices
- Index compression:
  - delta-encoding: convert big numbers to small ones
  - v-byte encoding: use fewer bits for small numbers
  - skip pointers: random access into compressed lists

Summary (2)
- Index construction
  - naive: for each term in doc, update inv list
    - core idea: construct partial indices, then merge
    - distributed indexing: MapReduce
    - distributed search: execute in parallel, merge

- Query execution
  - term-at-a-time: fast, memory-intensive
  - doc-at-a-time: flexible, disk-intensive
  - caching: only most frequent queries
  - heuristics: try to skip some documents