Search Engine Architecture

4. Indexing and Retrieval
The Central Problem in Search

Do these represent the same concepts?

Abstract IR Architecture

Query Representation → Comparison Function → Hits

Documents → Representation Function → Index

How do we represent text?

• Remember: computers don’t “understand” anything!
• “Bag of words”
  • Treat all the words in a document as index terms
  • Assign a “weight” to each term based on “importance” (or, in simplest case, presence/absence of word)
  • Disregard order, structure, meaning, etc. of the words
  • Simple, yet effective!
• Assumptions
  • Term occurrence is independent
  • Document relevance is independent
  • “Words” are well-defined

Counting Words...

documents → bag of words → inverted index

case folding, tokenization, stopword removal, stemming

syntax, semantics, word knowledge, etc.

Boolean Retrieval

- Users express queries as a Boolean expression
  - AND, OR, NOT
  - Can be arbitrarily nested
- Retrieval is based on the notion of sets
  - Any given query divides the collection into two sets: retrieved, not-retrieved
  - Pure Boolean systems do not define an ordering of the results

Inverted Index: Boolean Retrieval

Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham

Hybrid of a KV store and a column store?

Boolean Retrieval

• To execute a Boolean query:
  • Build query syntax tree
  
  \[(\text{blue AND fish}) \text{ OR ham}\]

• For each clause, look up postings
  
  \[
  \begin{array}{c}
  \text{blue} \rightarrow 2 \\
  \text{fish} \rightarrow 1 \rightarrow 2
  \end{array}
  \]

• Traverse postings and apply Boolean operator

• Efficiency analysis
  • Postings traversal is linear (assuming sorted postings)
  • Start with shortest posting first

Strengths and Weaknesses

• **Strengths**
  • Precise, if you know the right strategies
  • Precise, if you have an idea of what you’re looking for
  • Implementations are fast and efficient

• **Weaknesses**
  • Users must learn Boolean logic
  • Boolean logic insufficient to capture the richness of language
  • No control over size of result set: either too many hits or none
  • **When do you stop reading?** All documents in the result set are considered “equally good”
  • **What about partial matches?** Documents that “don’t quite match” the query may be useful also

Ranked Retrieval

- Order documents by how likely they are to be relevant
  - Estimate relevance($q, d_i$)
  - Sort documents by relevance
  - Display sorted results
- User model
  - Present hits one screen at a time, best results first
  - At any point, users can decide to stop looking
- How do we estimate relevance?
  - Assume document is relevant if it has a lot of query terms
  - Replace relevance($q, d_i$) with sim($q, d_i$)
  - Compute similarity of vector representations

Assumption: Documents that are “close together” in vector space “talk about” the same things.

Therefore, retrieve documents based on how close the document is to the query (i.e., similarity ~ “closeness”)

Term Weighting

• Term weights consist of two components
  • Local: how important is the term in this document?
  • Global: how important is the term in the collection?
• Here’s the intuition:
  • Terms that appear often in a document should get high weights
  • Terms that appear in many documents should get low weights
• How do we capture this mathematically?
  • Term frequency (local)
  • Inverse document frequency (global)

TF.IDF Term Weighting

\[ w_{i,j} = tf_{i,j} \cdot \log \frac{N}{n_i} \]

- \( w_{i,j} \) weight assigned to term \( i \) in document \( j \)
- \( tf_{i,j} \) number of occurrence of term \( i \) in document \( j \)
- \( N \) number of documents in entire collection
- \( n_i \) number of documents with term \( i \)

Inverted Index: TF.IDF

Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham

Positional Indexes

- Store term position in postings
- Supports richer queries (e.g., proximity)
- Naturally, leads to larger indexes...

Inverted Index: Positional Information

Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham

\[ tf \]

\[ df \]

blue 1 1 1 1 1

red 1 1 1 1 1

two 1 1 1 1 1

Retrieval in a Nutshell

• Look up postings lists corresponding to query terms
• Traverse postings for each query term
• Store partial query-document scores in accumulators
• Select top $k$ results to return

Retrieval: Document-at-a-Time

- Evaluate documents one at a time (score all query terms)
  - Document score in top k?
    - Yes: Insert document score, extract-min if queue too large
    - No: Do nothing

- Tradeoffs
  - Small memory footprint (good)
  - Must read through all postings (bad), but skipping possible
  - More disk seeks (bad), but blocking possible

Retrieval: Query-At-A-Time

- Evaluate documents one query term at a time
  - Usually, starting from most rare term (often with tf-sorted postings)

\[
\text{Score}_{(q=x)}(\text{doc } n) = s \rightarrow \text{Accumulators (e.g., hash)}
\]

- Tradeoffs
  - Early termination heuristics (good)
  - Large memory footprint (bad), but filtering heuristics possible

Indexing vs. Retrieval

- The indexing problem
  - Scalability is critical
  - Must be relatively fast, but need not be real time
  - Fundamentally a batch operation
  - Incremental updates may or may not be important
  - For the web, crawling is a challenge in itself

- The retrieval problem
  - Must have sub-second response time
  - For the web, only need relatively few results

Postings Encoding

Conceptually:

fish 1 2 9 1 21 3 34 1 35 2 80 3 ...

In Practice:

- Don’t encode docnos, encode gaps (or $d$-gaps)
- But it’s not obvious that this saves space...

Overview of Index Compression

- Byte-aligned vs. bit-aligned
- Byte-aligned technique
  - VByte
  - Simple9 and variants
  - PForDelta
- Non-parameterized bit-aligned
  - Unary codes
  - γ codes
  - δ codes
- Parameterized bit-aligned
  - Golomb codes (local Bernoulli model)

Want more detail? Read *Managing Gigabytes* by Witten, Moffat, and Bell!

VByte

- Simple idea: use only as many bytes as needed
  - Need to reserve one bit per byte as the “continuation bit”
  - Use remaining bits for encoding value

7 bits: 0 1 0 1 0 1 0
14 bits: 1 0 0 1 0 1 0 1 0 1 0 1 0 1
21 bits: 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

- Works okay, easy to implement...

Unary Codes

- $x \geq 1$ is coded as $x-1$ one bits, followed by 1 zero bit
  - $3 = 110$
  - $4 = 1110$
- Great for small numbers... horrible for large numbers
  - Overly-biased for very small gaps

Watch out! Slightly different definitions in different textbooks

γ codes

• $x \geq 1$ is coded in two parts: length and offset
  • Start with binary encoded, remove highest-order bit = offset
  • Length is number of binary digits, encoded in unary code
  • Concatenate length + offset codes
• Example: 9 in binary is 1001
  • Offset = 001
  • Length = 4, in unary code = 1110
  • $\gamma$ code = 1110:001
• Analysis
  • Offset = $\lceil \log x \rceil$
  • Length = $\lceil \log x \rceil + 1$
  • Total = $2 \lceil \log x \rceil + 1$

\(\delta\) codes

- Similar to \(\gamma\) codes, except that length is encoded in \(\gamma\) code
- Example: 9 in binary is 1001
  - Offset = 001
  - Length = 4, in \(\gamma\) code = 11000
  - \(\delta\) code = 11000:001
- \(\gamma\) codes = more compact for smaller numbers
- \(\delta\) codes = more compact for larger numbers

Golomb Codes

- \( x \geq 1 \), parameter \( b \):
  - \( q + 1 \) in unary, where \( q = \lfloor \frac{x - 1}{b} \rfloor \)
  - \( r \) in binary, where \( r = x - qb - 1 \), in \( \lceil \log b \rceil \) or \( \lfloor \log b \rfloor \) bits

- Example:
  - \( b = 3, r = 0, 1, 2 \) (0, 10, 11)
  - \( b = 6, r = 0, 1, 2, 3, 4, 5 \) (00, 01, 100, 101, 110, 111)
  - \( x = 9, b = 3: q = 2, r = 2, \text{ code } = 110:11 \)
  - \( x = 9, b = 6: q = 1, r = 2, \text{ code } = 10:100 \)

- Optimal \( b \approx 0.69 \) (N/df)
  - Different \( b \) for every term!

## Comparison of Coding Schemes

<table>
<thead>
<tr>
<th>Unary</th>
<th>γ</th>
<th>δ</th>
<th>Golomb b=3</th>
<th>Golomb b=6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0:0</td>
<td>0:00</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>10:0</td>
<td>100:0</td>
<td>0:10</td>
</tr>
<tr>
<td>3</td>
<td>110</td>
<td>10:1</td>
<td>100:1</td>
<td>0:11</td>
</tr>
<tr>
<td>4</td>
<td>1110</td>
<td>110:00</td>
<td>101:00</td>
<td>10:0</td>
</tr>
<tr>
<td>5</td>
<td>11110</td>
<td>110:01</td>
<td>101:01</td>
<td>10:10</td>
</tr>
<tr>
<td>6</td>
<td>111110</td>
<td>110:10</td>
<td>101:10</td>
<td>10:11</td>
</tr>
<tr>
<td>7</td>
<td>1111110</td>
<td>110:11</td>
<td>101:11</td>
<td>110:0</td>
</tr>
<tr>
<td>8</td>
<td>11111110</td>
<td>110:000</td>
<td>11000:000</td>
<td>110:10</td>
</tr>
<tr>
<td>9</td>
<td>111111110</td>
<td>110:001</td>
<td>11000:001</td>
<td>110:11</td>
</tr>
<tr>
<td>10</td>
<td>1111111110</td>
<td>110:010</td>
<td>11000:010</td>
<td>1110:0</td>
</tr>
</tbody>
</table>

# Index Compression: Performance

## Comparison of Index Size (bits per pointer)

<table>
<thead>
<tr>
<th></th>
<th>Bible</th>
<th>TREC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary</td>
<td>262</td>
<td>1918</td>
</tr>
<tr>
<td>Binary</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>6.51</td>
<td>6.63</td>
</tr>
<tr>
<td>$\delta$</td>
<td>6.23</td>
<td>6.38</td>
</tr>
<tr>
<td>Golomb</td>
<td>6.09</td>
<td>5.84</td>
</tr>
</tbody>
</table>

*One of the best techniques*

Bible: King James version of the Bible; 31,101 verses (4.3 MB)
TREC: TREC disks 1+2; 741,856 docs (2070 MB)
Term vs. Document Partitioning

Document Partitioning

- Requires query broker
- Lower query latencies
- Hot spots easier to manage
- Supports multi-phase search strategy
  - Start with highest-quality documents, ...
Questions?