Introduction to Information Retrieval and Boolean model

Reference: Introduction to Information Retrieval by C. Manning, P. Raghavan, H. Schutze
Unstructured (text) vs. structured (database) data in late nineties
Unstructured (text) vs. structured (database) data now
Goal of IR

- Collection: A set of documents
- Goal: Find documents relevant to user’s information need and helps the user complete a task
Boolean Model for IR

• Queries are Boolean expressions.
  – e.g., Caesar AND Brutus

• The search engine returns all documents that satisfy the Boolean expression.
Boolean queries: Exact match

• Queries using **AND**, **OR** and **NOT** together with query terms
  – Views each document as a **set** of words
  – Is precise: document matches condition or not.

• Primary commercial retrieval tool for 3 decades.

• Professional searchers (e.g., Lawyers) still like Boolean queries:
  – You know exactly what you’re getting.
Example: WestLaw

- Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992)

  - About 7 terabytes of data; 700,000 users
  - Majority of users still use Boolean queries

http://legalsolutions.thomsonreuters.com/law-products/westlaw-legal-research/
Example: WestLaw

• Example query:
  – What is the statute of limitations in cases involving the federal tort claims act?
  – LIMIT! /3 STATUTE ACTION /S FEDERAL /2 TORT /3 CLAIM

• Long, precise queries; proximity operators; incrementally developed; not like web search
Retrieval for Shakespeare Document Collection

• Which plays of Shakespeare contain the words *Brutus AND Caesar* but *NOT Calpurnia*?

• Could `grep` all of Shakespeare’s plays for *Brutus* and *Caesar*, then strip out lines containing *Calpurnia*?
  – Slow (for large corpora)
  – *NOT Calpurnia* is non-trivial
  – Other operations (e.g., find the phrase *Romans and countrymen*) not feasible
## Term-document incidence

**Query:** Brutus AND Caesar but NOT Calpurnia

<table>
<thead>
<tr>
<th></th>
<th>Antony and Cleopatra</th>
<th>Julius Caesar</th>
<th>The Tempest</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Antony</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Brutus</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Caesar</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cleopatra</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mercy</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>worser</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

1 if document contains word, 0 otherwise
Incidence vectors

- So we have a 0/1 vector for each term.
- To answer query: take the vectors for Brutus, Caesar and Calpurnia (complemented) → bitwise AND.
- 110100 AND 110111 AND 101111 = 100100.
Answers to query

• Antony and Cleopatra, Act III, Scene ii

Agrippa [Aside to DOMITIUS ENOBARBUS]: Why, Enobarbus,
When Antony found Julius Caesar dead,
He cried almost to roaring; and he wept
When at Philippi he found Brutus slain.

• Hamlet, Act III, Scene ii

Lord Polonius: I did enact Julius Caesar I was killed i’ the Capitol; Brutus killed me.
Bigger document collections

- Consider $N = 1$million documents, each with about 1K terms.
- Avg 6 bytes/term incl spaces/punctuation
  - 6GB of data in the documents.
- Say there are $M = 500K$ distinct terms among these.
Can’t build the matrix

• 500K x 1M matrix has half-a-trillion 0’s and 1’s.
• But it has no more than one billion 1’s.
  – matrix is extremely sparse.
• What’s a better representation?
  – We only record the 1 positions.
Inverted index

• For each term $T$: store a list of all documents that contain $T$.
• Each document is identified by a document ID

Brutus

Calpurnia

Caesar

What happens if the word *Caesar* is added to document 14?
Inverted index

• Use a variable-sized posting lists
  – Dynamic space allocation
  – Insertion of terms into documents easy
  – In memory, can use linked lists

Sorted by docID (more later on why)
Inverted index construction

Documents to be indexed

Token stream

More on these later.

Modified tokens

Inverted index

Tokenizer

Linguistic modules

Indexer

Friends, Romans, countrymen

friend

roman

countryman

friend

roman

countryman
Indexer steps

- Sequence of (Modified token, Document ID) pairs.

Doc 1

I did enact Julius Caesar I was killed i' the Capitol; Brutus killed me.

Doc 2

So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious

Term | Doc #
--- | ---
I | 1
did | 1
enact | 1
julius | 1
caesar | 1
I | 1
was | 1
killed | 1
i' | 1
the | 1
capitol | 1
brutus | 1
killed | 1
me | 1
so | 2
let | 2
it | 2
be | 2
with | 2
caesar | 2
the | 2
noble | 2
brutus | 2
hath | 2
told | 2
you | 2
caesar | 2
was | 2
ambitious | 2
Indexer steps

• Sort by terms.

Core indexing step
Indexer steps

- Multiple term entries in a single document are merged.
- Frequency information is added.

Why frequency? Will discuss later.
The result is split into a Dictionary file and a Postings file.
Query processing

• Consider processing the query:

\[ \text{Brutus AND Caesar} \]

– Locate \textit{Brutus} in the Dictionary;
  • Retrieve its postings.

– Locate \textit{Caesar} in the Dictionary;
  • Retrieve its postings.

– “Merge” the two postings:

\[
\begin{array}{cccccccc}
2 & 4 & 8 & 16 & 32 & 64 & 128 \\
1 & 2 & 3 & 5 & 8 & 13 & 21 & 34
\end{array}
\]
The merge

- Walk through the two postings simultaneously, in time linear in the total number of postings entries.

If the list lengths are $x$ and $y$, the merge takes $O(x+y)$ operations.

**Crucial**: postings sorted by docID.
Basic postings intersection

• A “merge” algorithm

INTERSECT\( (p_1, p_2) \)
1 \( answer \leftarrow \langle \rangle \)
2 \textbf{while} \( p_1 \neq \text{NIL and } p_2 \neq \text{NIL} \)
3 \textbf{do if} \( \text{docID}(p_1) = \text{docID}(p_2) \)
4 \quad \textbf{then} \( \text{ADD}(answer, \text{docID}(p_1)) \)
5 \quad \quad p_1 \leftarrow \text{next}(p_1) \)
6 \quad p_2 \leftarrow \text{next}(p_2) \)
7 \textbf{else if} \( \text{docID}(p_1) < \text{docID}(p_2) \)
8 \quad \textbf{then} \( p_1 \leftarrow \text{next}(p_1) \)
9 \quad \textbf{else} \( p_2 \leftarrow \text{next}(p_2) \)
10 \textbf{return} \( answer \)
Query optimization

- What is the best order for query processing?
- Consider a query that is an AND of $t$ terms.
- For each of the $t$ terms, get its postings, then AND together.

```
Brutus  →  2 4 8 16 32 64 128
Calpurnia →  1 2 3 5 8 16 21 34
Caesar   →  13 16
```

Query: Brutus AND Calpurnia AND Caesar
Query optimization example

- Process in order of increasing freq:
  - *start with smallest set, then keep cutting further.*

This is why we kept freq in dictionary

Brutus

Calpurnia

Caesar

Execute the query as (*Caesar AND Brutus*) AND *Calpurnia*.
INTERSECT(⟨t₁, . . . , tₙ⟩)
1 terms ← SORTBYINCREASINGFREQUENCY(⟨t₁, . . . , tₙ⟩)
2 result ← POSTINGS(FIRST(terms))
3 terms ← REST(terms)
4 while terms ≠ NIL and result ≠ NIL
5 do list ← POSTINGS(FIRST(terms))
6 result ← INTERSECT(result, POSTINGS(FIRST(terms)))
7 terms ← REST(terms)
8
9 return result

Figure 1.8 Algorithm for conjunctive queries that returns the set of documents containing each term in the input list of terms.
More general optimization

• e.g., \( \textit{madding OR crowd} \) AND \( \textit{ignoble OR strife} \)

• Get freq’s for all terms.

• Estimate the size of each \( OR \) by the sum of its freq’s (conservative).

• Process in increasing order of \( OR \) sizes.
Phrase queries

• We want to be able to answer queries such as “air conditioner” – as a phrase
• Thus the sentence “After washing my hair with this conditioner, I dry my hair with hot air” is not a match.
  – The concept of phrase queries has proven easily understood by users; one of the few “advanced search” ideas that works
  – Many more queries are implicit phrase queries
• For this, it no longer suffices to store only 
  <term : docs> entries
A first attempt: Biword indexes

• Index every consecutive pair of terms in the text as a phrase
• For example the text “Friends, Romans, Countrymen” would generate the biwords
  – friends romans
  – romans countrymen
• Each of these biwords is now a dictionary term
• Two-word phrase query-processing is now immediate.
Longer phrase queries

- Longer phrases can be processed by breaking them down
- *air conditioner filter system* can be broken into the Boolean query on biwords:
  \[ \text{air conditioner AND conditioner filter AND filter system} \]

Without the docs, we cannot verify that the docs matching the above Boolean query do contain the phrase.

Can have false positives!
Issues for biword indexes

• False positives, as noted before
• Index blowup due to bigger dictionary
  – Infeasible for more than biwords, big even for them

• Biword indexes are not the standard solution (for all biwords) but can be part of a compound strategy
Solution 2: Positional indexes

• In the postings, store, for each term the position(s) in which tokens of it appear:

<term, number of docs containing term; doc1: position1, position2 ... ;
doc2: position1, position2 ... ; etc.>
Positional index example

<be: 993427;
  1: 7, 18, 33, 72, 86, 231;
  2: 3, 149;
  4: 17, 191, 291, 430, 434;
  5: 363, 367, ...>

- For phrase queries, we use a merge algorithm recursively at the document level
- But we now need to deal with more than just equality

Which of docs 1, 2, 4, 5 could contain “to be or not to be”?
Processing a phrase query

• Extract inverted index entries for each distinct term: *to, be, or, not.*

• Merge their *doc:position* lists to enumerate all positions with “*to be or not to be*”.
  
  — *to*:
  
  • 2:1,17,74,222,551; 4:8,16,190,429,433; 7:13,23,191; ...
  
  — *be*:
  
  • 1:17,19; 4:17,191,291,430,434; 5:14,19,101; ...

• Same general method for proximity searches
Proximity queries

• **LIMIT! /3 STATUTE /3 FEDERAL /2 TORT**
  – Again, here, /k means “within k words of”.
  
• Clearly, positional indexes can be used for such queries; biword indexes cannot.
Positional index size

• A positional index expands postings storage substantially
  – Even though indices can be compressed
• Nevertheless, a positional index is now standardly used because of the power and usefulness of phrase and proximity queries ... whether used explicitly or implicitly in a ranking retrieval system.
Rules of thumb

• A positional index is 2–4 as large as a non-positional index

• Positional index size 35–50% of volume of original text

  – Caveat: all of this holds for “English-like” languages
Combination schemes

• These two approaches can be profitably combined
  – For particular phrases ("Michael Jackson", "Britney Spears") it is inefficient to keep on merging positional postings lists
    • Even more so for phrases like "The Who"

• Williams et al. (2004) evaluate a more sophisticated mixed indexing scheme
  – A typical web query mixture was executed in ¼ of the time of using just a positional index
  – It required 26% more space than having a positional index alone
Structured vs unstructured data

- Structured data tends to refer to information in “tables”

<table>
<thead>
<tr>
<th>Employee</th>
<th>Manager</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>Jones</td>
<td>50000</td>
</tr>
<tr>
<td>Chang</td>
<td>Smith</td>
<td>60000</td>
</tr>
<tr>
<td>Ivy</td>
<td>Smith</td>
<td>50000</td>
</tr>
</tbody>
</table>

Typically allows numerical range and exact match (for text) queries, e.g.,

\[ Salary < 60000 \text{ AND Manager} = \text{Smith}. \]
Unstructured data

• Typically refers to free text
• Allows
  – Keyword queries including operators
  – More sophisticated “concept” queries e.g.,
    • find all web pages dealing with *drug abuse*
• Classic model for searching text documents
Semi-structured data

• But in fact almost no data is “unstructured”
• E.g., this slide has distinctly identified zones such as the *Title* and *Bullets*
• Facilitates “semi-structured” search such as
  – *Title* contains **data** AND *Bullets* contain **search**