AI6122 Text Data Management & Analysis

Topic: Boolean Retrieval

Outline

- Inverted index
- Processing Boolean queries
- Query optimization
- Phrase query
  - Biword index
  - Positional index

Information Retrieval

- Information Retrieval (IR) is **finding material** (usually documents) of an **unstructured nature** (usually text) that satisfies an **information need** from within **large collections** (usually stored on computers).

- Related definitions
  - **Information need:** The topic about which the user desires to know more
  - **Query**: What the user conveys to the computer in an attempt to communicate the information need
  - **Relevant document**: user perceives as containing information of value with respect to the information need

Boolean Retrieval

- The Boolean model is arguably the simplest model to base an information retrieval system on queries are Boolean expressions
  - Example: Brutus **AND** Caesar

- The search engine return all documents that satisfy the Boolean expression
  - [without ranking ?]

Example: unstructured data in 1680

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

- One could **grep** all of Shakespeare's plays for ***Brutus*** and ***Caesar***, then strip out lines containing ***Calpurnia***?

- Why is "grep" is not the solution? [unix command]
  - Slow (for large corpora)
  - grep is line oriented, IR is document-oriented
  - NOT Calpurnia is non-trivial
  - Other operations (e.g., find the word **Romans** near **countrymen**) not feasible

Inverted index

- For each term t, we store **a list of all documents that contain t**
  - Each document is identified by a unique **docID**

Brutus	→	1, 2, 4, 11, 31, 45, 173, 174
Caesar	→	1, 2, 4, 5, 6, 16, 57, 132
Calpurnia	→	2, 31, 54, 101

posting

Dictionary

postings

Inverted index construction

Documents to be indexed: Friends, Romans, countrymen.

Tokenizer

Token stream: Friends | Romans | countrymen

Linguistic modules

Modified tokens: friend | roman | countrymen

Indexer

Inverted index:

- friend → 2, 4, …
- roman → 1, 2, …
- countrymen → 13, 26, …

Indexer steps: Token sequence

- 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	docID
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

- Sort by terms, and then docID

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

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

Term	docID
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

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Indexer steps: Dictionary & Postings

- Multiple term entries in a single document are merged.
- Split into Dictionary and Postings
- Document frequency information is added.

Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

→

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
i	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

Where do we pay in storage?

Terms and counts

term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
i	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

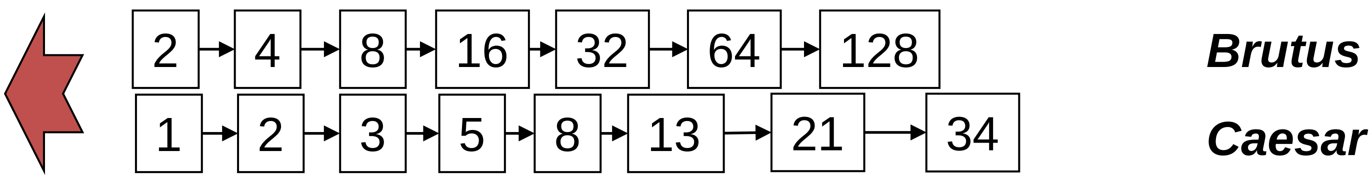
Lists of docIDs

Later in the course:

- How do we index efficiently?
- How much storage do we need?

Query processing: AND

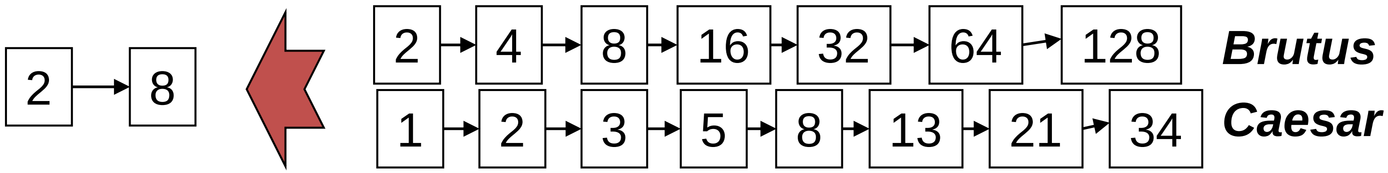
- Consider processing the query: ***Brutus*** *AND* ***Caesar***
  - Locate ***Brutus*** in the Dictionary;
    - Retrieve its postings.
  - Locate ***Caesar*** in the Dictionary;
    - Retrieve its postings.
  - "Merge" the two postings:



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.
  - Assumption: postings sorted by docID.



Intersecting two postings lists (a "merge" algorithm)

```
INTERSECT(p1, p2)
answer ← ⟨ ⟩
while p1 ≠ NIL and p2 ≠ NIL
do if docID(p1) = docID(p2)
     then ADD(answer, docID(p1))
          p1 ← next(p1)
          p2 ← next(p2)
     else if docID(p1) < docID(p2)
            then p1 ← next(p1)
            else p2 ← next(p2)
return answer
```

Boolean queries: Exact match

- The **Boolean retrieval model** is being able to ask a query that is a Boolean expression:
  – Boolean Queries are queries using AND, OR and NOT to join query terms
    - Views each document as a set of words
    - Is precise: document matches condition or not.
  – Perhaps the **simplest model** to build an IR system on

- Primary commercial retrieval tool for 3 decades.

- Many search systems you still use are Boolean:
  – Email, library catalog, Mac OS X Spotlight

Example: WestLaw   http://www.westlaw.com/

- Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992)
  - Tens of terabytes of data; 700,000 users
  - Majority of users still use Boolean queries

- 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
    - /3 = within 3 words, /S = in same sentence

- Long, precise queries; proximity operators; incrementally developed; not like web search
  - Many professional searchers still like Boolean search
  - You know exactly what you are getting

Query optimization

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

Query: ***Brutus*** *AND* ***Calpurnia*** *AND* ***Caesar***

Brutus	⟶		2, 4, 8, 16, 32, 64, 128
Caesar	⟶		1, 3, 4, 8, 16, 21, 34
Calpurnia	⟶		13, 16

Query optimization example

- Process in order of increasing frequency:
  - start with smallest set, then keep cutting further.
  - This is why we kept document frequency in dictionary

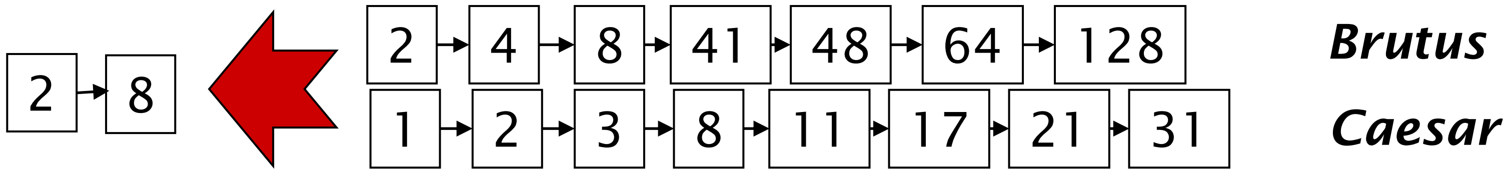
Brutus	→		2, 4, 8, 16, 32, 64, 128
Caesar	→		1, 3, 4, 8, 16, 21, 34
Calpurnia	→		13, 16

Execute the query as (***Calpurnia*** *AND* ***Brutus)*** *AND* ***Caesar***.

- For query like: **(madding OR crowd) AND (ignoble OR strife)**
  - Get document frequencies for all terms.
  - Estimate the size of each OR by the sum of its document frequencies (conservative).
  - Process in increasing order of OR sizes.

Re-look at the merging of postings

- Walk through the two postings simultaneously, in time linear in the total number of postings entries
  - If the list lengths are m and n, the merge takes $O(m+n)$ operations.

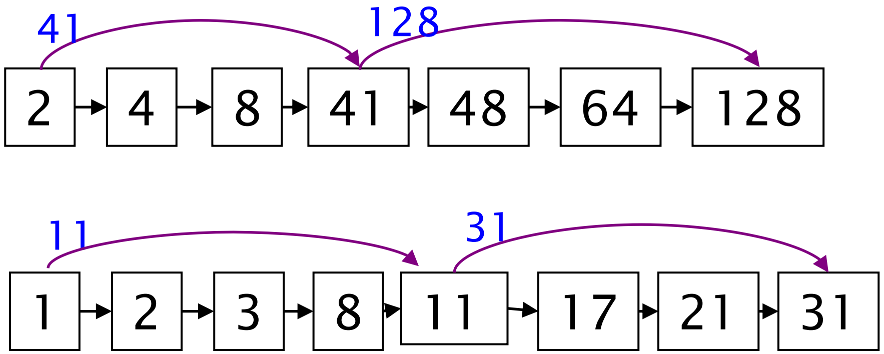


- Can we do better?
  - Assuming index does not change too fast

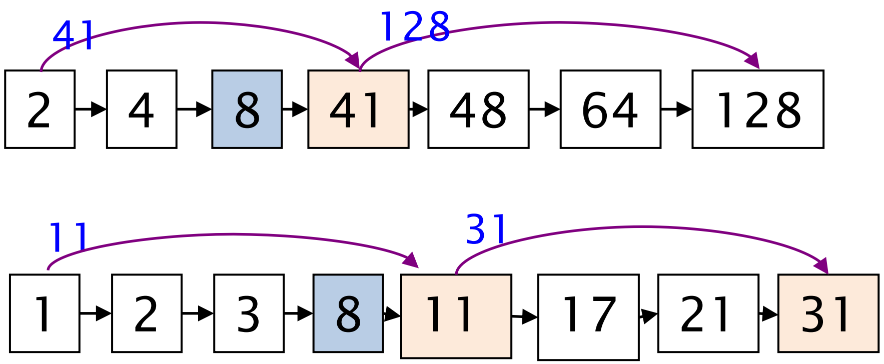
Augment postings with skip pointers (at indexing time)

- Why?
  – To skip postings that will not figure in the search results.

- How?
  – Where do we place skip pointers?



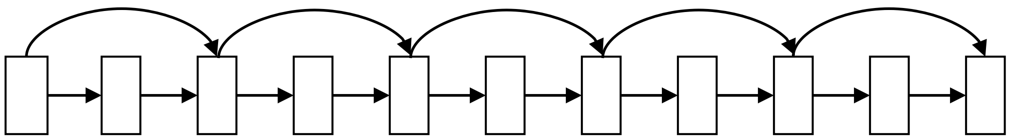
Query processing with skip pointers



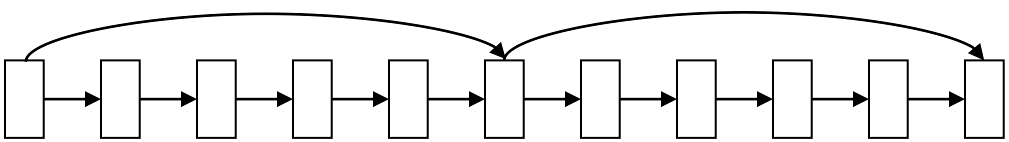
- Example
  - Suppose we’ve stepped through the lists until we process **8** on each list. We match it and advance.
  - We then have **41** and **11** on the lower. **11** is smaller.
  - But the skip successor of **11** on the lower list is **31**, so we can skip ahead past the intervening postings.

Where do we place skips? A tradeoff

- More skips $\rightarrow$ shorter skip spans $\Rightarrow$ more likely to skip.
  - But lots of comparisons to skip pointers.



- Fewer skips $\rightarrow$ few pointer comparison,
  - but then long skip spans $\Rightarrow$ few successful skips.



Placing skips

- Simple **heuristic**: for postings of length L, use $\sqrt{L}$ evenly-spaced skip pointers.
  - This ignores the distribution of query terms.
  - Easy if the index is relatively static;
  - Harder if L keeps changing because of updates.

- Remark:
  - This definitely **used** to help; with modern hardware it may not unless memory-based
  - The I/O cost of loading a bigger postings list can outweigh the gains from quicker in memory merging

Phrase queries

- Want to be able to answer queries such as "*stanford university*" – **as a phrase**

- Thus the sentence "I went to *university* at *Stanford*" 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 <u>implicit</u> phrase queries

- For this, it no longer suffices to store only

  <term : docs> entries

A first (not so good) attempt: Biword indexes

- Index every consecutive pair of terms in the text as a phrase

- For example: the text “Friends, Romans, Countrymen” would generate two 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: "***stanford university palo alto***" can be broken into the Boolean query on biwords:
  - ***stanford university*** *AND* ***university palo*** *AND* ***palo alto***

- Without the docs, we cannot verify that the docs matching the above Boolean query do contain the phrase.
  - Can have false positives

Extended biwords

- Parse the indexed text and perform part-of-speech-tagging. Bucket the terms into, for example
  - Nouns (N), Articles or prepositions (X).
  - Call any string of terms of the form NX*N an extended biword.
  - Each such extended biword is now a term in the dictionary.
  - Example: **catcher** in the **rye**
    N X X N

- Query processing: parse it into N's and X's
  - Segment query into enhanced biwords
  - Look up in index: **catcher rye**

Issues for biword indexes

- False positives

- Index blowup due to bigger dictionary
  - Infeasible for more than biwords, big even for them

- Biword indexes are not the standard solution, but can be **part of a compound strategy**

Positional indexes: a better solution

- 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.>

<***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

Processing a phrase query: “ to be or not to be”

- 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; ...

- Clearly, positional indexes can be used for such queries; biword indexes cannot.

Positional index size

- You can compress position values/offsets
  - Nevertheless, a positional index expands postings storage *substantially*
  - A positional index is 2–4 as large as a non-positional index
  - Positional index size 35–50% of volume of original text
  - all of this holds for "English-like" languages

- 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.

- Biword indexing can be **part of a compound strategy**
  - For particular phrases ("**Michael Jackson**", **"The Who")** it is inefficient to keep on merging positional postings lists
  - How about "**Britney Spears**"?

Summary

- Inverted index
- Processing Boolean queries
- Query optimization
- Phrase query
  - Biword index
  - Positional index