Introduction to

Information Retrieval

Lecture 1: Boolean retrieval
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 that not the answer?
  - Slow (for large corpora)
  - *NOT Calpurnia* is non-trivial
  - Other operations (e.g., find the word *Romans* near *countrymen*) not feasible
  - Ranked retrieval (best documents to return)
    - Later lectures
**Term-document incidence**

<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>

**Brutus AND Caesar BUT NOT Calpurnia**

1 if play 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.
- $110100 \text{ AND } 110111 \text{ AND } 101111 = 100100$. 
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.
Basic assumptions of Information Retrieval

- **Collection**: Fixed set of documents
- **Goal**: Retrieve documents with information that is **relevant** to the user’s **information need** and helps the user complete a **task**
The classic search model

- **TASK**: Get rid of mice in a politically correct way
- **Info Need**: Info about removing mice without killing them
- **Verbal form**: How do I trap mice alive?
- **Misconception?**
- **Mistranslation?**
- **Misformulation?**

**Query**: Get rid of mice in a politically correct way

**SEARCH ENGINE**:

- **Results**: Find this: `mouse trap` any language

**Corpus**:
How good are the retrieved docs?

- **Precision**: Fraction of retrieved docs that are relevant to user’s information need
- **Recall**: Fraction of relevant docs in collection that are retrieved
- More precise definitions and measurements to follow in later lectures
Bigger collections

- Consider $N = 1$ million documents, each with about 1000 words.
- Avg 6 bytes/word including 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.

Why?
Inverted index

- For each term $t$, we must store a list of all documents that contain $t$.
  - Identify each by a docID, a document serial number
- Can we use fixed-size arrays for this?

**Brutus**

1 2 4 11 31 45 173 174

**Caesar**

1 2 4 5 6 16 57 132

**Calpurnia**

2 31 54 101

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

- We need variable-size postings lists
  - On disk, a continuous run of postings is normal and best
  - In memory, can use linked lists or variable length arrays
    - Some tradeoffs in size/ease of insertion

*Dictionary*

Brutus

Caesar

Calpurnia

*Postings*

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.

Friends

Romans

Countrymen

Modified tokens:

friend

roman

countryman

Inverted index:

friend

roman

countryman

Documents to be indexed.

Tokenizer

Linguistic modules

Indexer

More on these later.

Modified tokens.

Inverted index.

Sec. 1.2
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.
Indexer steps: Sort

- Sort by terms
  - And then docID

Core indexing step

<table>
<thead>
<tr>
<th>Term</th>
<th>docID</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>I'</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
</tr>
<tr>
<td>so</td>
<td>2</td>
</tr>
<tr>
<td>let</td>
<td>2</td>
</tr>
<tr>
<td>it</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
</tr>
<tr>
<td>with</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
</tr>
<tr>
<td>noble</td>
<td>2</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
</tr>
<tr>
<td>hath</td>
<td>2</td>
</tr>
<tr>
<td>told</td>
<td>2</td>
</tr>
<tr>
<td>you</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
</tr>
<tr>
<td>ambitious</td>
<td>2</td>
</tr>
<tr>
<td>ambitious</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
</tr>
<tr>
<td>brutus</td>
<td>1</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
</tr>
<tr>
<td>hath</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>I'</td>
<td>1</td>
</tr>
<tr>
<td>it</td>
<td>2</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
</tr>
<tr>
<td>let</td>
<td>2</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
</tr>
<tr>
<td>noble</td>
<td>2</td>
</tr>
<tr>
<td>so</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
</tr>
<tr>
<td>told</td>
<td>2</td>
</tr>
<tr>
<td>you</td>
<td>2</td>
</tr>
<tr>
<td>was</td>
<td>1</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
</tr>
<tr>
<td>with</td>
<td>2</td>
</tr>
</tbody>
</table>
Indexer steps: Dictionary & Postings

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

Why frequency? Will discuss later.
Where do we pay in storage?

<table>
<thead>
<tr>
<th>term</th>
<th>doc. freq.</th>
<th>postings lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>ambitious</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>be</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>brutus</td>
<td>2</td>
<td>1, 1</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>hath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>i'</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>it</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>julius</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>killed</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>let</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>noble</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>so</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>told</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>you</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>was</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>with</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Pointers

Terms and counts

Lists of docIDs
More on Inverted Index Construction

- Hashing-based construction methods are more efficient (though harder to implement)
- Inverted index can be compressed to reduce index size
  - reduces transfer time between storage and memory
The index we just built

- How do we process a query?
  - Later - what kinds of queries can we process?

Today’s focus
Query processing: AND

- Consider processing the query:

  \textit{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.
Intersecting two postings lists (a “merge” algorithm)

\[
\text{INTERSECT}(p_1, p_2)
\]

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

- 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`
    - `foo! = foo*`, `/3 = within 3 words`, `/S = in same sentence`
Example: WestLaw  
http://www.westlaw.com/

- Another example query:
  - Requirements for disabled people to be able to access a workplace
  - disable! /p access! /s work-site work-place (employment /3 place

- Note that SPACE is disjunction, not conjunction!
- 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
- But that doesn’t mean it actually works better....
Boolean queries:
More general merges

- **Exercise**: Adapt the merge for the queries:

  - Brutus AND NOT Caesar
  - Brutus OR NOT Caesar

Can we still run through the merge in time $O(x+y)$?
What can we achieve?
Merging

What about an arbitrary Boolean formula?

\((\text{Brutus OR Caesar}) \ AND \ NOT \ (\text{Antony OR Cleopatra})\)

- Can we always merge in “linear” time?
  - Linear in what?
- Can we do better?
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*
Query optimization example

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

This is why we kept document freq. in dictionary

<table>
<thead>
<tr>
<th>Name</th>
<th>Frequencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brutus</td>
<td>2 4 8 16 32 64 128</td>
</tr>
<tr>
<td>Caesar</td>
<td>1 2 3 5 8 16 21 34</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>13 16</td>
</tr>
</tbody>
</table>

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

- e.g., \((madding \ OR \ crowd) \ AND \ (ignoble \ OR \ strife) \ AND \ (light \ OR \ lord)\)
- Get doc. freq.’s for all terms.
- **Estimate** the size of each *OR* by the sum of its doc. freq.’s (conservative).
- Process in increasing order of *OR* sizes.
Exercise

- Recommend a query processing order for

\[(\text{tangerine OR trees}) \ AND \ (\text{marmalade OR skies}) \ AND \ (\text{kaleidoscope OR eyes})]\n
<table>
<thead>
<tr>
<th>Term</th>
<th>Freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>eyes</td>
<td>213312</td>
</tr>
<tr>
<td>kaleidoscope</td>
<td>87009</td>
</tr>
<tr>
<td>marmalade</td>
<td>107913</td>
</tr>
<tr>
<td>skies</td>
<td>271658</td>
</tr>
<tr>
<td>tangerine</td>
<td>46653</td>
</tr>
<tr>
<td>trees</td>
<td>316812</td>
</tr>
</tbody>
</table>

Q: Any more accurate way to estimate the cardinality of intermediate results?

Q: Can we merge multiple lists (>2) simultaneously?
FASTER POSTINGS MERGES:
SKIP POINTERS/SKIP LISTS
Recall basic merge

- 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?
Yes (if index isn’t changing 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

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?

- **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 $L^{1/2}$ 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.

- This definitely used to help; with modern hardware it may not (Bahle et al. 2002) unless you’re memory-based
  - The I/O cost of loading a bigger postings list can outweigh the gains from quicker in memory merging!
Skip Pointers

- A skip pointer \((d, p)\) contains a document number \(d\) and a byte (or bit) position \(p\)
  - Means there is an inverted list posting that starts at position \(p\), and the posting before it was for document \(d\)
PHRASE QUERIES AND POSITIONAL INDEXES
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 implicit phrase queries
- For this, it no longer suffices to store only <term : docs> entries
Solution 1: 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 are processed as we did with wild-cards:
- *stanford university palo alto* can be broken into the Boolean query on biwords:

  \[
  \text{stanford university AND university palo AND palo alto}
  \]

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 (POST).
- Bucket the terms into (say) Nouns (N) and articles/prepositions (X).
- Call any string of terms of the form NX*N an extended biword.
  - Each such extended biword is now made 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, 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.
- Exercise: Adapt the linear merge of postings to handle proximity queries. Can you make it work for any value of k?
  - This is a little tricky to do correctly and efficiently
  - See Figure 2.12 of IIR (Page 39)
  - There’s likely to be a problem on it!
\textsc{PositionalIntersect}(p_1, p_2, k)
\begin{align*}
1 & \quad \text{answer} \leftarrow \langle \rangle \\
2 & \quad \textbf{while } p_1 \neq \text{NIL} \text{ and } p_2 \neq \text{NIL} \\
3 & \quad \textbf{do if } \text{docID}(p_1) = \text{docID}(p_2) \\
4 & \quad \quad \text{then } l \leftarrow \langle \rangle \\
5 & \quad \quad \text{pp}_1 \leftarrow \text{positions}(p_1) \\
6 & \quad \quad \text{pp}_2 \leftarrow \text{positions}(p_2) \\
7 & \quad \textbf{while } \text{pp}_1 \neq \text{NIL} \\
8 & \quad \quad \textbf{do while } \text{pp}_2 \neq \text{NIL} \\
9 & \quad \quad \quad \textbf{do if } |\text{pos(pp}_1) - \text{pos(pp}_2)| \leq k \\
10 & \quad \quad \quad \quad \text{then } \text{ADD}(l, \text{pos(pp}_2)) \\
11 & \quad \quad \quad \quad \text{else if } \text{pos(pp}_2) > \text{pos(pp}_1) \\
12 & \quad \quad \quad \quad \quad \text{then break} \\
13 & \quad \quad \quad \quad \quad \text{pp}_2 \leftarrow \text{next(pp}_2) \\
14 & \quad \quad \quad \textbf{while } l \neq \langle \rangle \text{ and } |l[0] - \text{pos(pp}_1)| > k \\
15 & \quad \quad \quad \textbf{do } \text{DELETE}(l[0]) \\
16 & \quad \quad \quad \textbf{for each } ps \in l \\
17 & \quad \quad \quad \quad \textbf{do } \text{ADD}(\text{answer}, \langle \text{docID}(p_1), \text{pos(pp}_1), ps \rangle) \\
18 & \quad \quad \quad \quad \text{pp}_1 \leftarrow \text{next(pp}_1) \\
19 & \quad \quad \quad \quad \quad \text{pp}_2 \leftarrow \text{next(pp}_2) \\
20 & \quad \textbf{else if } \text{docID}(p_1) < \text{docID}(p_2) \\
21 & \quad \quad \text{then } p_1 \leftarrow \text{next}(p_1) \\
22 & \quad \quad \quad \text{else } p_2 \leftarrow \text{next}(p_2) \\
23 & \quad \textbf{return } \text{answer}
\end{align*}
Positional index size

- You can compress position values/offsets: we’ll talk about that in lecture 5
- Nevertheless, a positional index expands postings storage substantially
- 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.
Positional index size

- Need an entry for each occurrence, not just once per document.
- Index size depends on average document size:
  - Average web page has <1000 terms.
  - SEC filings, books, even some epic poems ... easily 100,000 terms.
- Consider a term with frequency 0.1%.

<table>
<thead>
<tr>
<th>Document size</th>
<th>Postings</th>
<th>Positional postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>100,000</td>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>
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
Solution 3: Suffix Tree/Array

- **BANANA$**
  - **BANANA$** pos: 0
  - **ANANA$** pos: 1
  - **NANA$** pos: 2
  - **ANA$** pos: 3
  - **NA$** pos: 4
  - **A$** pos: 5

Sort on the strings

- **A$** pos: 5
- **ANA$** pos: 3
- **ANANA$** pos: 1
- **BANANA$** pos: 0
- **NA$** pos: 4
- **NANA$** pos: 2

$ < \text{any char}$
### Suffix Array

- **Suffix Array**
  - **BANANA$**
    - BANANA$ pos: 0
    - ANANA$ pos: 1
    - NANA$ pos: 2
    - ANA$ pos: 3
    - NA$ pos: 4
    - A$ pos: 5
    - $ pos: 6
  - $ pos: 6
  - A$ pos: 5
  - ANA$ pos: 3
  - ANANAN pos: 1
  - BANANA$ pos: 0
  - NA$ pos: 4
  - NANA$ pos: 2

- If the original string is available, each suffix can be completely specified by the **index** of its first character.

- Index size (incl. data): $5n$

- **Sort on the strings**

- **Binary search (using offsets to fetch the ‘key’)**

- **Suffix array**

- **$ < any char**
Resources for today’s lecture

- *Introduction to Information Retrieval*, chapter 1
- Shakespeare:
  - Try the neat browse by keyword sequence feature!
- *Managing Gigabytes*, chapter 3.2
- *Modern Information Retrieval*, chapter 8.2
Resources for today’s lecture

- Skip Lists theory: Pugh (1990)
  - Multilevel skip lists give same $O(\log n)$ efficiency as trees
  