Search: the beginning

Nisheeth
Interdisciplinary area

Search

Information retrieval

NLP

Machine learning

Human factors
Outline

• Components
  – Crawling
  – Processing
  – Indexing
  – Retrieval
  – Evaluation

• Research areas
  – Text processing
    • Beyond bag-of-words representations
  – Retrieval algorithms
    • Context sensitivity
    • Personalization
    • Diversity and serendipity
  – Evaluation methods
    • Usability studies
    • Real-time tracking
Emphasis areas

• Text processing
  – Basic algorithms
  – Criteria for model selection
  – Data transformations

• Retrieval
  – Algorithms
  – Ranking schemes
  – Research foci

• Evaluation
  – Existing methods
  – Problems
  – Research foci
De-emphasized areas

- Search engine architecture
- Crawling
- Indexing
- Scalability concerns
- Privacy concerns
Resources

• Croft, Metzler & Strohman (ex-Google)
  – Search Engines: Information Retrieval in Practice (pdf on HCC webpage)

• Chapters 1,2,4,6,7,8

• Other research papers and books as we go along
The simplest possible search model

BOOLEAN SEARCH
The classic search model

Person

Need

Query

Query refinement

Search engine

Results

Collection
Boolean search

• Search queries always Boolean formulae
  – Later deal with natural language queries
• No uncertainty about corpus membership
  – Later deal with document clusters
• Search intent is known
  – Later deal with context, personalization
• Great example – Gmail search
• Reading material
  – Manning, Raghavan & Schutze, Intro to IR
  – Chapter 1
  – Available online
Boolean retrieval: 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 document corpus

• 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 matrices

The following table represents a term-document incidence matrix, where each row corresponds to a term, and each column corresponds to a document. The entries in the matrix are binary, with 1 indicating the presence of the term in the document, and 0 indicating absence.

<table>
<thead>
<tr>
<th>Term</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 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 **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 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.
Inverted index

• For each term $t$, we must store a list of all documents that contain $t$.
  – Identify each doc by a docID, a document serial number

• Can we used fixed-size arrays for this?

<table>
<thead>
<tr>
<th>Brutus</th>
<th>1 4 11 31 45 173 174</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caesar</td>
<td>1 2 4 5 6 16 57 132</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>2 31 54 101</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Dictionary</th>
<th>Postings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brutus</td>
<td>1 2 4 11 31 45 173 174</td>
</tr>
<tr>
<td>Caesar</td>
<td>1 2 4 5 6 16 57 132</td>
</tr>
<tr>
<td>Calpurnia</td>
<td>2 31 54 101</td>
</tr>
</tbody>
</table>
Inverted index construction

Documents to be indexed

Token stream

Modified tokens

Inverted index

Tokenizer

Linguistic modules

Indexer

Inverted index

Friends, Romans, countrymen.

friend

roman

countryman

2

4

1

2

13

16

Friends

Romans

Countrymen
Initial stages of text processing

- **Tokenization**
  - Cut character sequence into word tokens
    - Deal with "John’s", a state-of-the-art solution

- **Normalization**
  - Map text and query term to same form
    - You want *U.S.A.* and *USA* to match

- **Stemming**
  - We may wish different forms of a root to match
    - *authorize, authorization*

- **Stop words**
  - We may omit very common words (or not)
    - *the, a, to, of*
Indexer steps: Token sequence

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

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.

<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>
</tbody>
</table>
Indexer steps: Sort

- Sort by terms
  - And then docID
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?

Terms and counts

<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>2</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
<td>1</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>2</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>1</td>
</tr>
<tr>
<td>me</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>noble</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>so</td>
<td>1</td>
<td>1</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>2</td>
</tr>
<tr>
<td>with</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Lists of docIDs

IR system implementation
- How do we index efficiently?
- How much storage do we need?
Query processing

• How do we process a query?
  – Later - what kinds of queries can we process?
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 (intersect the document sets):

\begin{center}
\begin{tikzpicture}

\node at (0,0) (Brutus) {Brutus};
\node at (1,0) (Caesar) {Caesar};
\node at (0,1) (BrutusA) {2};
\node at (0,2) (BrutusB) {1};
\node at (1,1) (CaesarA) {4};
\node at (1,2) (CaesarB) {2};
\node at (2,1) (BrutusC) {8};
\node at (2,2) (BrutusD) {3};
\node at (3,1) (BrutusE) {16};
\node at (3,2) (BrutusF) {5};
\node at (4,1) (BrutusG) {32};
\node at (4,2) (BrutusH) {8};
\node at (5,1) (BrutusI) {64};
\node at (5,2) (BrutusJ) {13};
\node at (6,1) (BrutusK) {128};
\node at (6,2) (BrutusL) {21};
\node at (7,1) (BrutusM) {34};
\node at (7,2) (BrutusN) {};

\draw[->] (Brutus) -- (BrutusA);
\draw[->] (BrutusA) -- (BrutusB);
\draw[->] (BrutusB) -- (BrutusC);
\draw[->] (BrutusC) -- (BrutusD);
\draw[->] (BrutusD) -- (BrutusE);
\draw[->] (BrutusE) -- (BrutusF);
\draw[->] (BrutusF) -- (BrutusG);
\draw[->] (BrutusG) -- (BrutusH);
\draw[->] (BrutusH) -- (BrutusI);
\draw[->] (BrutusI) -- (BrutusJ);
\draw[->] (BrutusJ) -- (BrutusK);
\draw[->] (BrutusK) -- (BrutusL);
\draw[->] (BrutusL) -- (BrutusM);
\draw[->] (BrutusM) -- (BrutusN);
\draw[->] (Caesar) -- (CaesarA);
\draw[->] (CaesarA) -- (CaesarB);
\draw[->] (CaesarB) -- (CaesarC);
\draw[->] (CaesarC) -- (CaesarD);
\draw[->] (CaesarD) -- (CaesarE);
\draw[->] (CaesarE) -- (CaesarF);
\draw[->] (CaesarF) -- (CaesarG);
\draw[->] (CaesarG) -- (CaesarH);
\draw[->] (CaesarH) -- (CaesarI);
\draw[->] (CaesarI) -- (CaesarJ);
\draw[->] (CaesarJ) -- (CaesarK);
\draw[->] (CaesarK) -- (CaesarL);
\draw[->] (CaesarL) -- (CaesarM);
\draw[->] (CaesarM) -- (CaesarN);
\end{tikzpicture}
\end{center}
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)

\textbf{INTERSECT}(p_1, p_2)

1. \textit{answer} \leftarrow \langle \rangle
2. \textbf{while} \; p_1 \neq \text{NIL} \; \text{and} \; p_2 \neq \text{NIL}
3. \textbf{do if} \; \text{doclID}(p_1) = \text{doclID}(p_2)
   \hspace{1cm} \textbf{then ADD}(\textit{answer}, \text{doclID}(p_1))
   \hspace{1cm} p_1 \leftarrow \text{next}(p_1)
   \hspace{1cm} p_2 \leftarrow \text{next}(p_2)
4. \textbf{else if} \; \text{doclID}(p_1) < \text{doclID}(p_2)
   \hspace{1cm} \textbf{then} \; p_1 \leftarrow \text{next}(p_1)
   \hspace{1cm} \textbf{else} \; p_2 \leftarrow \text{next}(p_2)
5. \textbf{return} \; \textit{answer}
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?

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

• 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.

<table>
<thead>
<tr>
<th>Brutus</th>
<th>2 4 8 16 32 64 128</th>
</tr>
</thead>
<tbody>
<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>

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 document freq. in dictionary

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

• e.g., *(madding OR crowd) AND (ignoble OR strife)*

• 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.
Quick review

<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</td>
</tr>
<tr>
<td>capitol</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>caesar</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>enact</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>hath</td>
<td>1</td>
<td>2</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>2</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>2</td>
</tr>
</tbody>
</table>

Terms and counts

Lists of docIDs
Exercise

• Recommend a query processing order for

(tangerine OR trees) AND (marmalade OR skies) AND (kaleidoscope OR eyes)

• Which two terms should we process first?

<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>
More optimization: skip pointers

- 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 the 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?
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 $\sqrt{L}$ evenly-spaced skip pointers [Moffat and Zobel 1996]
• 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 unless you’re memory-based [Bahle et al. 2002]
  – The I/O cost of loading a bigger postings list can outweigh the gains from quicker in memory merging!
Phrase queries

• We 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
Naïve method: 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
• *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!
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:

\[<\text{term}, \text{number of docs containing } \text{term}; \text{doc1: position1, position2 } ... ; \text{doc2: position1, position2 } ... ; \text{etc.}>\]
Positional index example

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

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

• 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

• 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
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.
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
  – Novels ... 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
  - At this point you can start thinking about compressing the index itself
Proximity queries: example

• Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992; new federated search added 2010)

• 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
Example: WestLaw

• Another example query:
  – Requirements for disabled people to be able to access a workplace
  – disabl! /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....
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.
- 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
Combination schemes

• These two approaches can be combined
  – For particular phrases ("Michael Jackson", "Britney Spears") it is inefficient to keep on merging positional postings lists

• 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