Introduction

Boolean retrieval

Inverted index

Boolean queries

Text processing

Phrase queries

Proximity search

NPFL103: Information Retrieval (1)
Introduction, Boolean retrieval, Inverted index, Text processing

Pavel Pecina
pecina@ufal.mff.cuni.cz
Lecturer

Institute of Formal and Applied Linguistics
Faculty of Mathematics and Physics
Charles University

Based on slides by Hinrich Schütze, University of Stuttgart.
Introduction
Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (stored on computers).
Boolean retrieval
Arguably the simplest model to base an IR system on

Based on Boolean logic and set theory

Documents to be searched are conceived as sets of terms

Queries are Boolean expressions, e.g., CAESAR AND BRUTUS

The system returns all documents that satisfy the Boolean expression.

Does Google use the Boolean model?
Does Google use the Boolean model?

- On Google, the default interpretation of a query \([w_1 \ w_2 \ldots \ w_n]\) is
  \[
  w_1 \ \text{AND} \ w_2 \ \text{AND} \ \ldots \ \text{AND} \ w_n
  \]

- Cases where you get hits that do not contain one of the \(w_i\):  
  - anchor text (<a href="http://web">anchor text</a>)  
  - page contains variant of \(w_i\) (morphology, spelling, synonymy)  
  - long queries (\(n\) large)  
  - boolean expression generates very few hits

- Other operators supported: NOT (-), OR (|), ...

- **Simple Boolean vs. Ranking of result set**  
  - Simple Boolean retrieval returns documents in no particular order.  
  - Google (and most well designed Boolean engines) rank the result set – good hits ranked higher than bad hits (according to some estimator of relevance).
Inverted index
Unstructured data in 1650: Plays of William Shakespeare
Unstructured data in 1650

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 not the solution?

- Slow (for large collections)
- grep is line-oriented, IR is document-oriented
- “not Calpurnia” is non-trivial
- Other operations (e.g. search for Romans near country) infeasible
Term-document incidence matrix

<table>
<thead>
<tr>
<th></th>
<th>Anthony and Cleopatra</th>
<th>Julius Caesar</th>
<th>The Tempest</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ANTHONY</strong></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>BRUTUS</strong></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>CAESAR</strong></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>CALPURNIA</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>CLEOPATRA</strong></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>MERCY</strong></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td><strong>WORSER</strong></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Entry is 1 if term occurs. Example: CALPURNIA occurs in *Julius Caesar*.
Entry is 0 if term doesn’t occur. Example: CALPURNIA doesn’t occur in *The tempest*.
So we have a 0/1 vector for each term.

To answer the query **Brutus and Caesar and not Calpurnia**:

1. Take the vectors for **Brutus**, **Caesar**, and **Calpurnia**
   
   110100, 110111, 010000

2. Complement the vector of **Calpurnia**
   
   not 010000 = 101111

3. Do a (bitwise) **AND** on the three vectors:
   
   110100 AND 110111 AND 101111 = 100100
### BRUTUS AND CAESAR AND NOT CALPURNIA

<table>
<thead>
<tr>
<th>Anthony and Cleopatra</th>
<th>Caesar</th>
<th>The Tempest</th>
<th>Hamlet</th>
<th>Othello</th>
<th>Macbeth</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ANTHONY</strong></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td><strong>BRUTUS</strong></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>CAESAR</strong></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>CALPURNIA</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>NOT CALPURNIA</strong></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Cleopatra</strong></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>mercy</strong></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>worsER</strong></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>result:</strong></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Anthony 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 $N = 10^6$ documents, each with about 1000 tokens
  $\Rightarrow$ total of $10^9$ tokens

- On average 6 bytes per token, including spaces and punctuation
  $\Rightarrow$ size of document collection is about $6 \cdot 10^9 = 6$ GB

- Assume there are $M = 500,000$ distinct terms in the collection
  $\Rightarrow M = 500,000 \times 10^6 = \text{half a trillion 0s and 1s.}$

- But the matrix has no more than one billion 1s.
  $\Rightarrow$ Matrix is extremely sparse.

- What is a better representations?
  $\Rightarrow$ We only record the 1s.
For each term $t$, we store a list of all documents that contain $t$.

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

...
Inverted index construction

1. Collect the documents to be indexed:

   Friends, Romans, countrymen. So let it be with Caesar ...

2. Tokenize the text, turning each document into a list of tokens:

   Friends Romans countrymen So ...

3. Do linguistic preprocessing, producing a list of normalized tokens, which are the indexing terms: friend roman countryman so ...

4. Index the documents that each term occurs in by creating an inverted index, consisting of a dictionary and postings.
Tokenization and preprocessing

**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:

**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
### Generate postings, sort, create lists, determine document frequency

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

**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
Split the result into dictionary and postings file

<table>
<thead>
<tr>
<th>Name</th>
<th>Dictionary postings file</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>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

**dictionary**

**postings file**
Boolean queries
Consider the query: **Brutus AND Calpurnia**

To find all matching documents using inverted index:

1. Locate **Brutus** in the dictionary
2. Retrieve its postings list from the postings file
3. Locate **Calpurnia** in the dictionary
4. Retrieve its postings list from the postings file
5. Intersect the two postings lists
6. Return intersection to user
Intersecting two postings lists

**BRUTUS** → 1 → 2 → 4 → 11 → 31 → 45 → 173 → 174

**CALPURNIA** → 2 → 31 → 54 → 101

Intersection  ⇒  2 → 31

- This is linear in the length of the postings lists.
- Note: This only works if postings lists are sorted.
**Intersecting two postings lists**

**INTERSECT**($p_1$, $p_2$)

1. $answer ← ⟨⟩$
2. while $p_1 ≠ \text{NIL}$ and $p_2 ≠ \text{NIL}$
3. do if $\text{docID}(p_1) = \text{docID}(p_2)$
   4. then $\text{ADD}(answer, \text{docID}(p_1))$
   5. $p_1 ← \text{next}(p_1)$
   6. $p_2 ← \text{next}(p_2)$
4. else if $\text{docID}(p_1) < \text{docID}(p_2)$
   5. then $p_1 ← \text{next}(p_1)$
   6. else $p_2 ← \text{next}(p_2)$
10. return $answer$
Boolean queries

- Boolean model can answer any query that is a Boolean expression.
  - Boolean queries use AND, OR and NOT to join query terms.
  - Views each document as a set of terms.
  - Is precise: Document matches condition or not.

- Primary commercial retrieval tool for 3 decades

- Many professional searchers (e.g., lawyers) still like Boolean queries.
  - You know exactly what you are getting.
Text processing
So far: Simple Boolean retrieval system

Our assumptions were:

1. We know what a document is.
2. We can “machine-read” each document.

This can be complex in reality.
Parsing a document

- We need to deal with **format** and **language** of each document.
- What format is it in? pdf, word, excel, html etc.
- What language is it in?
- What character set is in use?
- Each of these is a classification problem
- Alternative: use heuristics
A single index usually contains terms of several languages.

- Sometimes a document or its components contain multiple languages/formats (e.g. French email with Spanish pdf attachment)

What is the document unit for indexing?

- A file?
- An email?
- An email with 5 attachments?
- A group of files (ppt or latex in HTML)?

Upshot: Answering the question “what is a document?” is not trivial and requires some design decisions.
Definitions

- **Word** – A delimited string of characters as it appears in the text.
- **Term** – A “normalized” word (morphology, spelling, etc.); an equivalence class of words.
- **Token** – An instance of a word or term occurring in a document.
- **Type** – The same as a term in most cases: an equivalence class of tokens.
Normalization

- Need to “normalize” terms in indexed text as well as query terms into the same form.

**Example:** We want to match *U.S.A.* and *USA*

- We most commonly implicitly define *equivalence classes* of terms.

- Alternatively: do asymmetric expansion
  - *window* → *window, windows*
  - *windows* → *Windows, windows*
  - *Windows* → *Windows* (no expansion)

- More powerful, but less efficient

- Why don’t you want to put *window, Window, windows, and Windows* in the same equivalence class?
Normalization and language detection interact.

Example:

- *PETER WILL NICHT MIT.* → MIT = mit
- *He got his PhD from MIT.* → MIT ≠ mit
Recall: Inverted index construction

Input: Friends, Romans, countrymen. So let it be with Caesar...

Output: friend roman countryman so ...

Each token is a candidate for a postings entry.

What are valid tokens to emit?
Exercises

► How many word tokens? How many word types?

Example 1: In June, the dog likes to chase the cat in the barn.

Example 2: Mr. O’Neill thinks that the boys’ stories about Chile’s capital aren’t amusing.

► ...tokenization is difficult – even in English.
Tokenization problems: One word or two? (or several)

- Hewlett-Packard
- State-of-the-art
- co-education
- the hold-him-back-and-drag-him-away maneuver
- data base
- San Francisco
- Los Angeles-based company
- cheap San Francisco-Los Angeles fares
- York University vs. New York University
Numbers

- 3/20/91
- 20/3/91
- Mar 20, 1991
- B-52
- 100.2.86.144
- (800) 234-2333
- 800.234.2333

Older IR systems may not index numbers ...
...but generally it’s a useful feature.
莎拉波娃现在居住在美国东南部的佛罗里达。今年4月9日，莎拉波娃在美国第一大城市纽约度过了18岁生日。生日派对上，莎拉波娃露出了甜美的微笑。
The two characters can be treated as one word meaning ‘monk’ or as a sequence of two words meaning ‘and’ and ‘still’.
Other cases of “no whitespace”

- Compounds in Dutch, German, Swedish
- Computerlinguistik → Computer + Linguistik
- Lebensversicherungsgesellschaftsangestellter
  → leben + versicherung + gesellschaft + angestellter
- Inuit: tusaatsiarunnanngittualuujunga (I can’t hear very well.)
- Other languages with segmentation difficulties: Finnish, Urdu ...
ノーベル平和賞を受賞したワンガリ・マータイさんが名誉会長を務めるMOTTAINAIキャンペーンの一環として、毎日新聞社とマガジンハウスは「私の、もったいない」を募集します。皆様が日ごろ「もったいない」と感じて実践していることや、それにまつわるエピソードを800字以内の文章にまとめ、簡単な写真、イラスト、図などを添えて10月20日までにお送りください。大賞受賞者には、50万円相当の旅行券とエコ製品2点の副賞が贈られます。

▶ 4 different “alphabets”:
  ▶ Chinese characters
  ▶ Hiragana syllabary for inflectional endings and function words
  ▶ Katakana syllabary for transcription of foreign words and other uses
  ▶ Latin

▶ No spaces (as in Chinese).

▶ End user can express query entirely in hiragana!
Introduction  Boolean retrieval  Inverted index  Boolean queries  Text processing  Phrase queries  Proximity search

Arabic script

un bā tīk
/kitābun/ ‘a book’
Arabic script: Bidirectionality

'Algeria achieved its independence in 1962 after 132 years of French occupation.'

‘استقلت الجزائر في سنة 1962 بعد 132 عاما من الاحتلال الفرنسي.’

Bidirectionality is not a problem if text is coded in Unicode.
Accents and diacritics

- Accents: résumé vs. resume (simple omission of accent)
- Umlauts: Universität vs. Universitaet (substitution “ä” and “ae”)
- Most important criterion: How are users likely to write their queries for these words?
- Even in languages that standardly have accents, users often do not type them (e.g. Czech)
Case folding

- Reduce all letters to lower case

- Possible exceptions: capitalized words in mid-sentence

  Example: MIT vs. mit, Fed vs. fed

- It’s often best to lowercase everything since users will use lowercase regardless of correct capitalization.
Stop words

- stop words = extremely common words which would appear to be of little value in helping select documents matching a user need

- **Examples:** *a, an, and, are, as, at, be, by, for, from, has, he, in, is, it, its, of, on, that, the, to, was, were, will, with*

- Stop word elimination used to be standard in older IR systems.

- But you need stop words for phrase queries, e.g. “King of Denmark”

- Most web search engines index stop words.
More equivalence classing

- Soundex: phonetic equivalence, e.g. *Muller* = *Mueller*
- Thesauri: semantic equivalence, e.g. *car* = *automobile*
Lemmatization

- Reduce inflectional/variant forms to base form

- Examples:
  - *am, are, is* → *be*
  - *car, cars, car’s, cars’* → *car*
  - *the boy’s cars are different colors* → *the boy car be different color*

- Lemmatization implies doing “proper” reduction to dictionary headword form (the lemma).

- Two types:
  - inflectional (*cutting* → *cut*)
  - derivational (*destruction* → *destroy*)
Stemming

- Crude heuristic process that **chops off the ends of words** in the hope of achieving what “principled” lemmatization attempts to do with a lot of linguistic knowledge.

- Language dependent

- Often inflectional **and** derivational

- **Example** (derivational): *automate, automatic, automation* all reduce to *automat*
Porter algorithm

- Most common algorithm for stemming English

- Designed in 1980 by Martin Porter, later versions support other languages (known as snowball)

- Results suggest that it is at least as good as other stemming options

- Conventions + 5 phases of reductions applied sequentially

- Each phase consists of a set of commands.

- **Sample command:** Delete final *ement* if what remains is longer than 1 character (replacement → replac, cement → cement)

- **Sample convention:** Of the rules in a compound command, select the one that applies to the longest suffix.
### Porter stemmer: A few rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSES</td>
<td>caresses</td>
</tr>
<tr>
<td>IES</td>
<td>ponies</td>
</tr>
<tr>
<td>SS</td>
<td>caress</td>
</tr>
<tr>
<td>S</td>
<td>cats</td>
</tr>
</tbody>
</table>

Rule Example

- SSES → SS → caress
- IES → I → poni
- SS → SS → caress
- S → cats → cat
Three stemmers: A comparison

**Sample text:** Such an *analysis* can reveal *features* that are not easily visible from the *variations* in the individual genes and can lead to a picture of expression that is more biologically *transparent* and accessible to interpretation.

**Porter stemmer:** such an *analysis* can reveal *feature* that are not easily visible from the *variation* in the individual gene and can lead to a picture of expression that is more biologically *transparent* and accessible to interpretation.

**Lovins stemmer:** such an *analysis* can reveal *feature* that are not easily visible from the *variation* in the individual gene and can lead to a picture of expression that is more biologically *transparent* and accessible to interpretation.

**Paice stemmer:** such an *analysis* can reveal *feature* that are not easily visible from the *variation* in the individual gene and can lead to a picture of expression that is more biologically *transparent* and accessible to interpretation.
Does stemming improve effectiveness?

- In general, stemming increases effectiveness for some queries, and decreases effectiveness for others.

- Queries where stemming is likely to help:
  - [TARTAN SWEATERS], [SIGHTSEEING TOUR SAN FRANCISCO]
  - equivalence classes: {sweater,sweaters}, {tour,tours}

- Queries where stemming hurts:
  - [OPERATIONAL RESEARCH], [OPERATING SYSTEM], [OPERATIVE DENTISTRY]
  - Porter Stemmer equivalence class oper contains all of operate, operating, operates, operation, operative, operatives, operational.
Phrase queries
Phrase queries

- We answer a query such as [STANFORD UNIVERSITY] – as a phrase.

- “The inventor Stanford Ovshinsky never went to university” → not a match

- The concept of phrase query has proven easily understood by users.

- About 10% of web queries are phrase queries.

- Consequence for inverted index:
  It no longer suffices to store docIDs in postings lists.

- Two ways of extending the inverted index:
  1. biword index
  2. positional index
Biword indexes

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

- **Example:** *Friends, Romans, Countrymen* generate two biwords: “friends romans” and “romans countrymen”

- Each of these biwords is now a vocabulary term.

- Two-word phrases can now easily be answered.
A long phrase like "stanford university palo alto" can be represented as the Boolean query "stanford university" AND "university palo" AND "palo alto"

We need to do post-filtering of hits to identify subset that actually contains the 4-word phrase.
Why are biword indexes rarely used?

- False positives, as noted above (post-filtering)
- Index blow-up due to very large term vocabulary
Positional indexes

- Positional indexes are a more efficient alternative to biword indexes.
- Postings lists in a nonpositional index: each posting is just a docID.
- Postings lists in a positional index: each posting is a docID and a list of positions.
**Positional indexes: Example**

**Query:** “\(to_1\) \(be_2\) \(or_3\) \(not_4\) \(to_5\) \(be_6\) ”

**TO, 993427:**
\[
\langle 1: \langle 7, 18, 33, 72, 86, 231 \rangle; \\
2: \langle 1, 17, 74, 222, 255 \rangle; \\
4: \langle 8, 16, 190, 429, 433 \rangle; \\
5: \langle 363, 367 \rangle; \\
7: \langle 13, 23, 191 \rangle; \ldots \rangle
\]

**BE, 178239:**
\[
\langle 1: \langle 17, 25 \rangle; \\
4: \langle 17, 191, 291, 430, 434 \rangle; \\
5: \langle 14, 19, 101 \rangle; \ldots \rangle
\]

**Document 4 is a match!**
Proximity search
We just saw how to use a positional index for phrase searches.

We can also use it for proximity search.

For example: `employment /4 place`

⇒ find all documents that contain `employment` and `place` within 4 words of each other.

“Employment agencies that place healthcare workers are seeing growth”
→ is a hit.

“Employment agencies that have learned to adapt now place healthcare workers” → is not a hit.
Proximity search

- Use the positional index

- Simplest algorithm: look at all combinations of positions of (i) EMPLOYMENT in document and (ii) PLACE in document

- Very inefficient for frequent words, especially stop words

- Note that we want to return the actual matching positions, not just a list of documents.

- This is important for dynamic summaries etc.
“Proximity” intersection

**POSITIONAL\textsc{INTERSECT}(p_1, p_2, k)**

1. \[ \text{answer} \leftarrow \langle \rangle \]
2. \[ \text{while } p_1 \neq \text{NIL} \text{ and } p_2 \neq \text{NIL} \]
3. \[ \text{do if } \text{docID}(p_1) = \text{docID}(p_2) \]
4. \[ \text{then } l \leftarrow \langle \rangle \]
5. \[ pp_1 \leftarrow \text{positions}(p_1) \]
6. \[ pp_2 \leftarrow \text{positions}(p_2) \]
7. \[ \text{while } pp_1 \neq \text{NIL} \]
8. \[ \text{do while } pp_2 \neq \text{NIL} \]
9. \[ \text{do if } |\text{pos}(pp_1) - \text{pos}(pp_2)| \leq k \]
10. \[ \text{then } \text{ADD}(l, \text{pos}(pp_2)) \]
11. \[ \text{else if } \text{pos}(pp_2) > \text{pos}(pp_1) \]
12. \[ \text{then break} \]
13. \[ pp_2 \leftarrow \text{next}(pp_2) \]
14. \[ \text{while } l \neq \langle \rangle \text{ and } |l[0] - \text{pos}(pp_1)| > k \]
15. \[ \text{do DELETE}(l[0]) \]
16. \[ \text{for each } ps \in l \]
17. \[ \text{do ADD}(\text{answer}, \langle \text{docID}(p_1), \text{pos}(pp_1), ps \rangle) \]
18. \[ pp_1 \leftarrow \text{next}(pp_1) \]
19. \[ p_1 \leftarrow \text{next}(p_1) \]
20. \[ p_2 \leftarrow \text{next}(p_2) \]
21. \[ \text{else if } \text{docID}(p_1) < \text{docID}(p_2) \]
22. \[ \text{then } p_1 \leftarrow \text{next}(p_1) \]
23. \[ \text{else } p_2 \leftarrow \text{next}(p_2) \]
24. \[ \text{return answer} \]
Combination scheme

- Biword indexes and positional indexes can be profitably combined.
- Many biwords extremely frequent: *Michael Jackson, Lady Gaga* etc.
- For these biwords, increased speed compared to positional postings intersection is substantial.
- Combination scheme: Include frequent biwords as vocabulary terms in the index. Do all other phrases by positional intersection.
- Williams et al. (2004) evaluate a more sophisticated mixed indexing scheme. Faster than a positional index, at a cost of 26% more space for index.
For web search engines, positional queries are much more expensive than regular Boolean queries.

Let’s look at the example of phrase queries.

Why are they more expensive than regular Boolean queries?

Can you demonstrate on Google that phrase queries are more expensive than Boolean queries?