

# Introduction to Information Retrieval

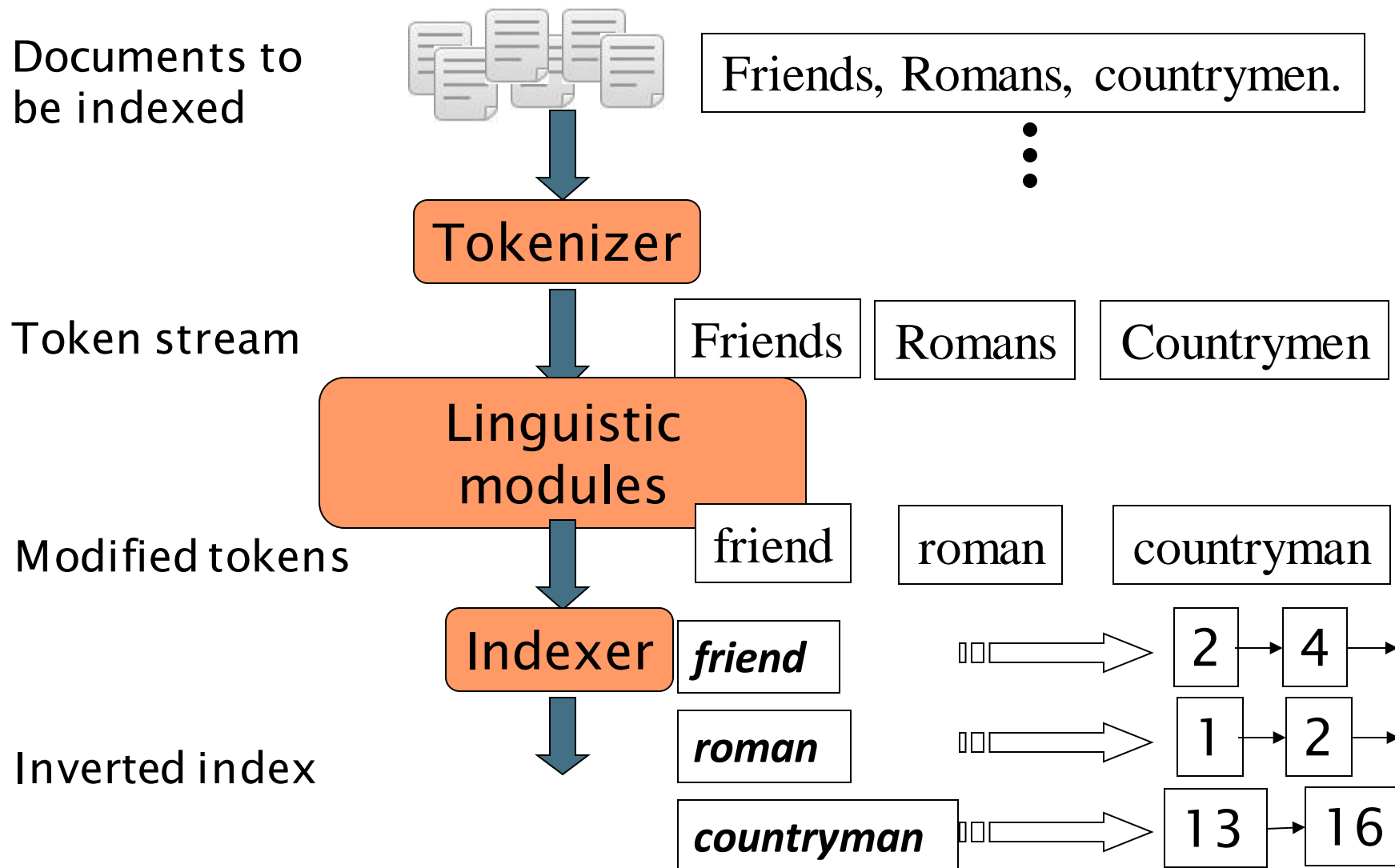
*The term vocabulary and postings  
lists*

# Contents

---

- **Tokenization**
- **Linguistic Preprocessing**
- **Faster postings list intersection via skip pointers**
- **Positional postings and phrase queries**

# Recall the basic indexing pipeline



# Major Steps in Inverted Index Construction

---

1. Collect the documents to be indexed.
2. Tokenize the text.
3. Do linguistic preprocessing of tokens.
4. Index the documents that each term occurs in.

# 1. Obtaining the character sequence in a document

---

- Digital documents that are the input to an indexing process
- We need to convert this byte sequence into sequence of characters
- **What Encoding Scheme?**
  - ASCII, Unicode UTF-8, vendor-specific standards
- **What format is it in?**
  - pdf/word/excel/html/zip?

- What language is it in?
  - Decoding Arabic, where text takes on some two dimensional and mixed order characteristics

ك ت ا ب ← كتاب  
 un b ā t i k  
 /kitābun/ 'a book'

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

← → ← → ← START

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

- Each of these is a classification problem.
- But these tasks are often handled by heuristic methods, user selection, or doc metadata
- There are commercial and open source libraries that can handle a lot of this stuff

## 2. Choosing a document unit (Indexing Granularity)

---

We return from our query “documents” but there are often interesting questions of grain size:

What is a unit document?

- An email with multiple attachments? // **Divide**
- A group of files (e.g., PPT or LaTeX split over HTML pages) // **Aggregate files**
- Very large document (book, chapter, paragraph) // **can be alleviated by use of proximity search**
- **An IR system should be designed to offer choices of granularity**

# Introduction to **Information Retrieval**

Tokenization



# 3. Tokenization

---

- Input: “*Friends, Romans and Countrymen*”
- Output: Tokens
  - *Friends*
  - *Romans*
  - *Countrymen*
- A **token** is an instance of a sequence of characters that are grouped together as a useful semantic unit
- Each such token type is now a candidate for an index entry (Term), after further processing
- **Type**: all tokens containing the same character sequence
- **Term**: normalized type that is included in the dictionary

# Issues in tokenization

---

- Starting point: you chop on **whitespace**
  - But what about
    - **Names:** *San Francisco*: one token or two?
    - **Phone numbers:** *(800) 234-2333*
    - **Dates:** *Mar. 12, 1991*
- Chop on all non-alphanumeric characters
  - But what about
    - **Apostrophe:** it is used for possession and contractions (e.g., boys' stories, aren't)
    - Email addresses ([ahmed@yahoo.com](mailto:ahmed@yahoo.com)), webURLs (<http://stuff.big.com/new/specials.html>), numeric IP addresses (142.32.48.231)

# Issues in tokenization

---

- ***Hyphen: used for***
  - splitting up vowels in words (*co-education*)
  - joining nouns (*Run-time*)
  - Word grouping (*hold-him-back*)
  - It can be effective to get the user to put in possible hyphens and the system generates all three forms (e.g., over-eager, over eager, overeager )
- No Spaces (language issues)
  - German noun compounds are not segmented
    - *E.g.: Lebensversicherungsgesellschaftsangestellter*

# Issues in tokenization

---

- Chinese and Japanese have no spaces between words:
  - 莎拉波娃现在居住在美国东南部的佛罗里达。
- Approaches to handle this issue
  - *word segmentation*: having a large vocabulary and taking the longest vocabulary match with some heuristics
  - Machine learning sequence models
  - K-grams

## 4. Dropping Stop words

---

- With a stop list, you exclude from the dictionary entirely the commonest words. Intuition:
  - They have little semantic content: *the, a, and, to, be*
  - There are a lot of them: ~30% of postings for top 30 words
- But the trend is away from doing this:
  - Good compression techniques (IIR 5) means the space for including stop words in a system is very small
  - Good query optimization techniques (IIR 7) mean you pay little at query time for including stop words.
  - You need them for:
    - Phrase queries: “King of Denmark”
    - Various song titles, etc.: “Let it be”, “To be or not to be”
    - “Relational” queries: “flights to London”

## 5. Normalization to terms

---

- Normalization is the process of canonicalizing tokens so that matches occur despite differences in the character sequences.
  - We want to match ***U.S.A.*** and ***USA***
  - ***Organize, organizes, and organizing***
- Result is terms: a **term** is a (normalized) word type, which is an entry in our IR system dictionary
- We most commonly implicitly define equivalence classes of terms by, **(i) using mapping rules (ii) list of synonyms**
  - deleting periods to form a term
    - ***U.S.A., USA***

# Normalization: other languages

---

- deleting hyphens to form a term
  - *anti-discriminatory, antidiscriminatory*
- Accents: e.g., naïve vs. naive.
  - Should be equivalent
  - Even in languages that standardly have accents, users often may not type them
  - Often best to normalize to a de-accented term
- Normalization of things like
  - May 30 vs. 5/30
  - Color vs. colour

# Case folding

---

- Reduce all letters to lower case
  - exception: upper case in mid-sentence?
    - E.g., companies (General Motors, The Associated Press)
    - Government organizations (the Fed vs. fed)
    - person names (Bush, Black).
  - Often best to lower case everything, since users will use lowercase regardless of 'correct' capitalization...



# Introduction to **Information Retrieval**

Stemming and Lemmatization

# Lemmatization

---

- Reduce inflectional/variant forms to base form
- E.g.,
  - *am, are, is* → *be*
  - *car, cars, car's, cars'* → *car*
  - *saw* → *see or saw*
- *the boy's cars are different colors* → *the boy car be different color*
- Lemmatization implies doing “proper” reduction via *Natural Language Processing*

# Stemming

---

- Reduce terms to their “roots” before indexing
- “Stemming” chops off the ends of words
  - language dependent
  - e.g., *automate(s), automatic, automation* all reduced to *automat*.

***for example compressed and compression are both accepted as equivalent to compress.***



for exampl compress and  
compress ar both accept  
as equival to compress

# Porter's algorithm

---

- Commonest algorithm for stemming English
  - Results suggest it's at least as good as other stemming options
- Conventions + 5 phases of reductions
  - phases applied sequentially
  - each phase consists of a set of commands
  - sample convention: *Of the rules in a compound command, select the one that applies to the longest suffix.*

# Typical rules in Porter

---

- | Rule                                 | Example           |
|--------------------------------------|-------------------|
| ■ SSES → SS                          | caresses → caress |
| ■ IES → I                            | ponies → poni     |
| ■ SS → SS                            | caress → caress   |
| ■ S →                                | cats → cat        |
| ■ Weight of word sensitive rules     |                   |
| ■ $(m > 1)$ <i>EMENT</i> →           |                   |
| ■ <i>replacement</i> → <i>replac</i> |                   |
| ■ <i>cement</i> → <i>cement</i>      |                   |

# Other stemmers

---

- Other stemmers exist:
  - Lovins stemmer
    - <http://www.comp.lancs.ac.uk/computing/research/stemming/general/lovins.htm>
    - Single-pass, longest suffix removal (about 250 rules)
  - Paice/Husk stemmer
  - Snowball

# Language-specificity

---

- The above methods embody transformations that are
  - Language-specific, and often
  - Application-specific
- These are “plug-in” addenda to the indexing process
- Both open source and commercial plug-ins are available for handling these

# Does stemming help?

---

- English: very mixed results. Helps recall for some queries but harms precision on others
  - E.g., *operate operating operates operation operative operatives operational*  $\Rightarrow$  oper
- Definitely useful for Spanish, German, Finnish, ...

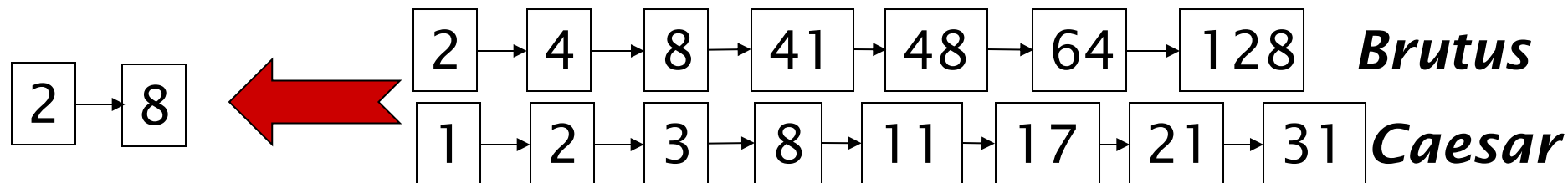


# Introduction to **Information Retrieval**

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

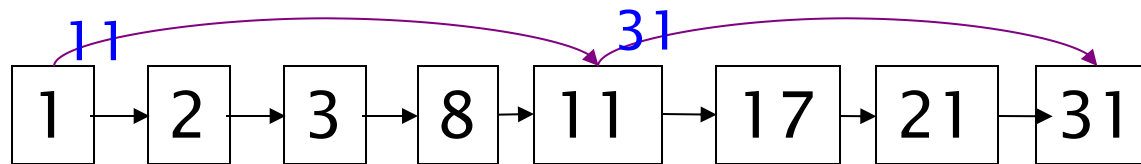
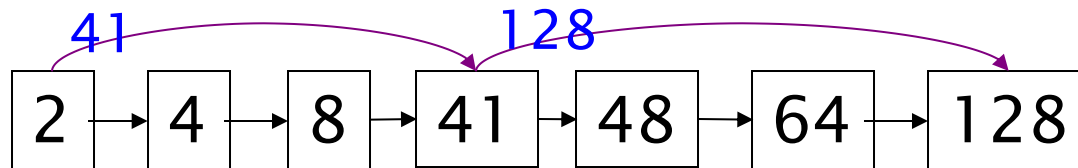


If the list lengths are  $m$  and  $n$ , the merge takes  $O(m+n)$  operations.

Can we do better?

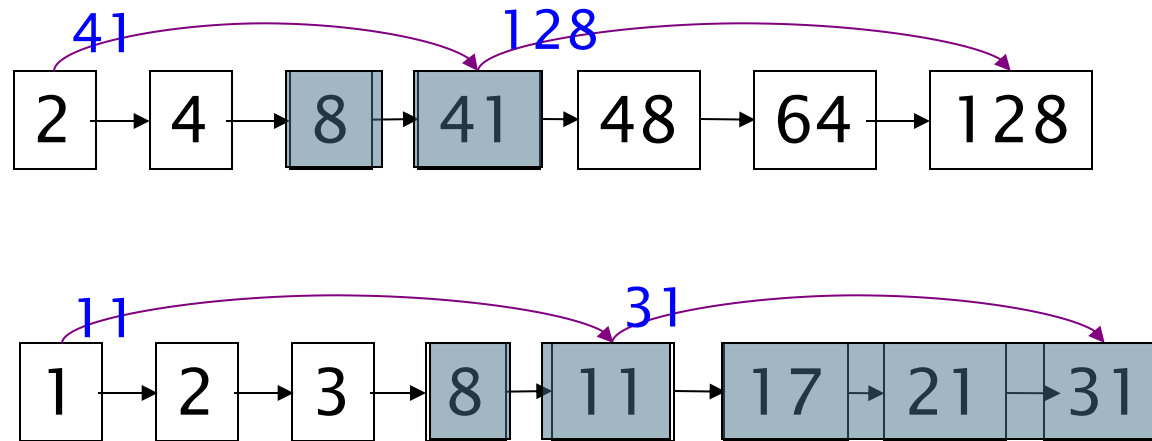
Yes (if the index isn't changing too fast).

# 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?

# 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.

# Postings lists intersection with skip pointers.

```

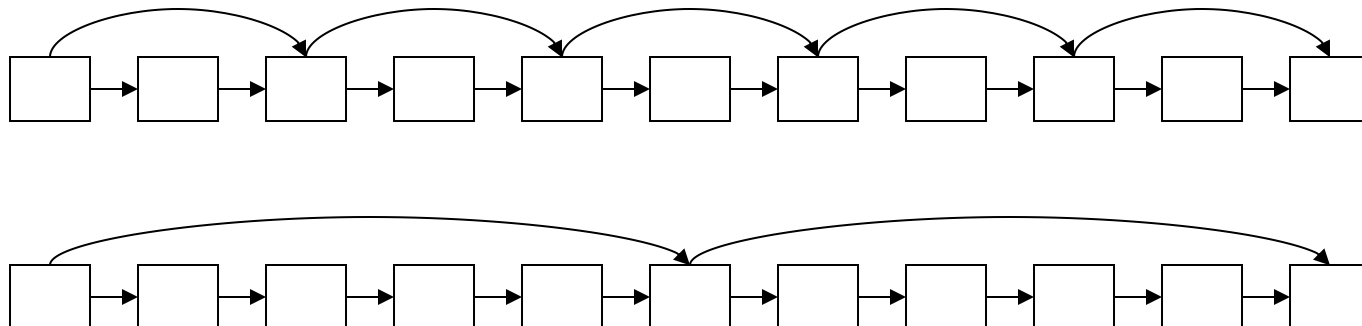
INTERSECTWITHSKIPS( $p_1, p_2$ )
1   $answer \leftarrow \{\}$ 
2  while  $p_1 \neq \text{NIL}$  and  $p_2 \neq \text{NIL}$ 
3  do if  $\text{docID}(p_1) = \text{docID}(p_2)$ 
4      then  $\text{ADD}(answer, \text{docID}(p_1))$ 
5           $p_1 \leftarrow \text{next}(p_1)$ 
6           $p_2 \leftarrow \text{next}(p_2)$ 
7  else if  $\text{docID}(p_1) < \text{docID}(p_2)$ 
8      then if  $\text{hasSkip}(p_1)$  and  $(\text{docID}(\text{skip}(p_1)) \leq \text{docID}(p_2))$ 
9          then while  $\text{hasSkip}(p_1)$  and  $(\text{docID}(\text{skip}(p_1)) \leq \text{docID}(p_2))$ 
10             do  $p_1 \leftarrow \text{skip}(p_1)$ 
11             else  $p_1 \leftarrow \text{next}(p_1)$ 
12  else if  $\text{hasSkip}(p_2)$  and  $(\text{docID}(\text{skip}(p_2)) \leq \text{docID}(p_1))$ 
13      then while  $\text{hasSkip}(p_2)$  and  $(\text{docID}(\text{skip}(p_2)) \leq \text{docID}(p_1))$ 
14          do  $p_2 \leftarrow \text{skip}(p_2)$ 
15          else  $p_2 \leftarrow \text{next}(p_2)$ 
16  return  $answer$ 

```

► **Figure 2.10** Postings lists intersection with skip pointers.

# 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!

# Introduction to **Information Retrieval**

Phrase queries and positional indexes



# 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

# 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
- ***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!

# 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
- The concept of a biword index can be extended to longer sequences of words
- If the index includes variable length word sequences, it is generally referred to as a ***phrase index***.

## 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

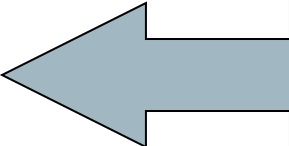
<*be*: 993427;

*1*: 7, 18, 33, 72, 86, 231;

*2*: 3, 149;

*4*: 17, 191, 291, 430, 434;

*5*: 363, 367, ...>



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

# 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*



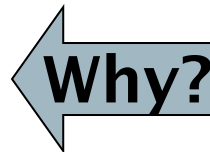
# Positional index size

---

- 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
  - SEC filings, books, even some epic poems ... easily 100,000 terms
- Consider a term with frequency 0.1%



Document size	Postings	Positional postings
1 000	1	1
100,000	1	100

# 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  $\frac{1}{4}$  of the time of using just a positional index
  - It required 26% more space than having a positional index alone