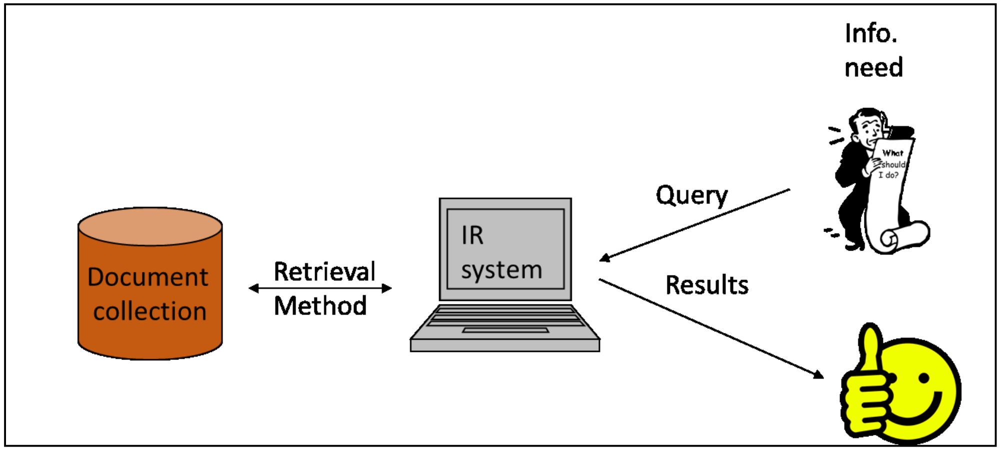
CS 429
Information Retrieval

Mid Term Review

The IR Problem

- **Goal** = find documents *relevant* to an information need from a large document set
  - Documents may be text, images, audio, etc.



- Typical IR Task
  - Inputs:
    - A corpus of documents (e.g. Text, images etc)
    - A query from the user in the form of a textual string
  - Output:
    - A (ranked) set of documents that are relevant to the query

Challenges of IR

- User Information Need
  - People have different and highly varied needs for information
  - People often do not know what they want, or may not be able to express it in a usable form
    Precision
  - How to satisfy these user needs for information?
- Dynamically changing content
  - E.g. sports web page displaying scores of Chicago Cubs game
  - E.g. facebook page
- Non-text formats
  - E.g. Images

Challenges of IR

- Evaluating Performance
  - Of the components
    - Indexing / matching algorithms
    - How accurate? -> Precision
    - How complete? -> Recall
  - Of the overall user experience
    - Usability issues
    - Usefulness to task
    - User satisfaction
- Visualization
  - Basic principles:
    - Overview first
    - Zoom
    - Details on demand

Text retrieval Problem

- First applications: in libraries (1950s)

  ISBN: 0-201-12227-8

  Author: Salton, Gerard

  Title: Automatic text processing: the transformation, analysis, and retrieval of information by computer

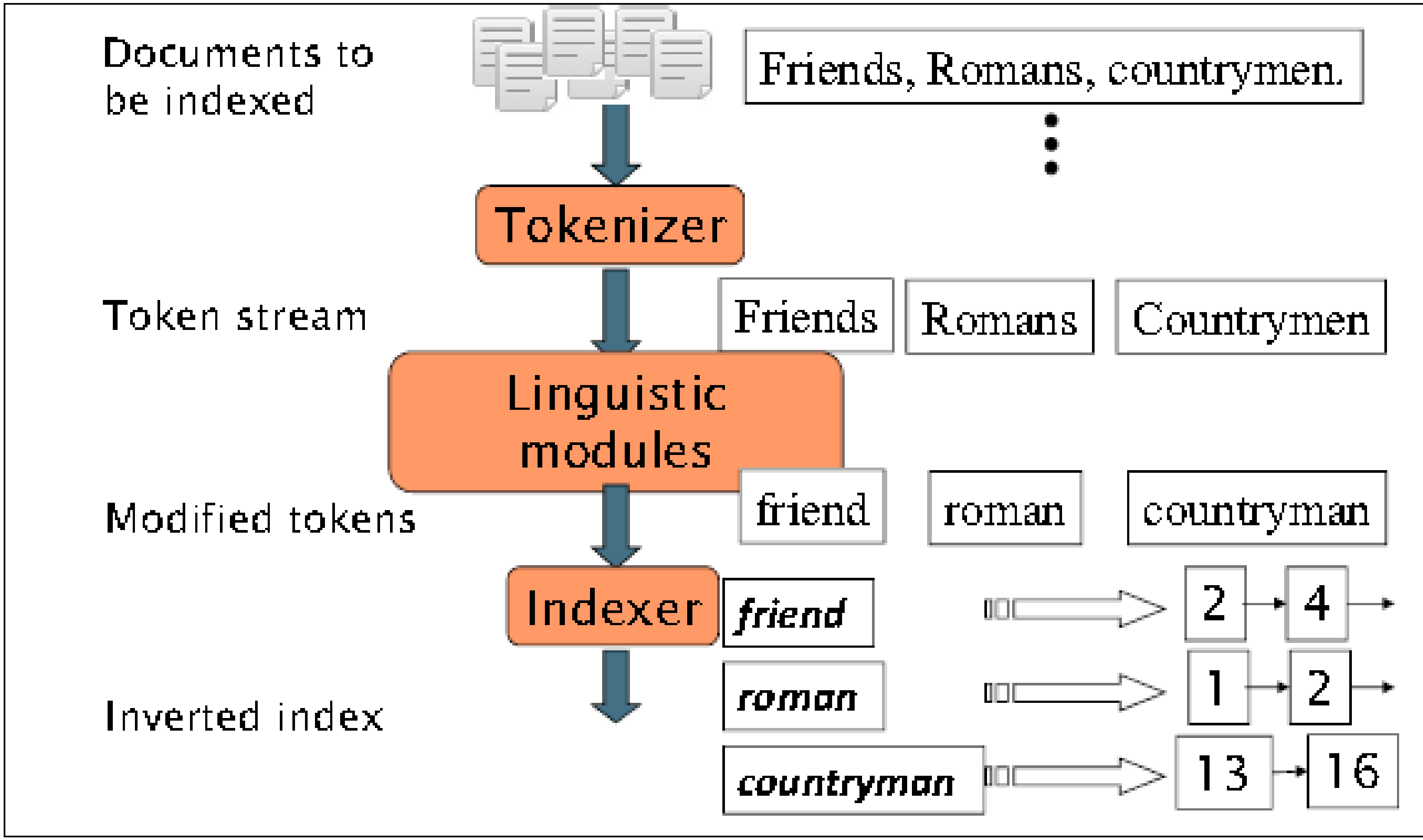
  Editor: Addison-Wesley

  Date: 1989

  Content: <Text>

- External attributes (structured documents) and internal attribute (content)
  - Search by external attributes = Search in DB (e.g. SQL)
  - Search by content

Indexing Pipeline



Text Processing Steps Prior to Indexing

- Tokenization
    - Extract word tokens from character sequence
- Remove stop words
    - Omit very common words
    - ***E.g. the, a, to, of***
- Normalization
    - Map types to terms
- Stemming
    - We may wish different forms of a root to match
    - ***E.g. “authorize” “authorization” has similar meaning***

Tokenization

- Some of the Issues in tokenization:
  - ***What to do with punctuation?***
    - *Finland's capital* → *Finland* AND *s*? *Finlands*? *Finland's*?
    - rosie o'donnell, 80's, 1890's, men's straw hats, master's degree,
  - ***Hyphenation***
    - *Hewlett-Packard* → *Hewlett* and *Packard* as two tokens?
    - *state-of-the-art*: break up hyphenated sequence.
    - *co-education*
    - *lowercase, lower-case, lower case ?*
  - *San Francisco*: one token or two?
    - How do you decide it is one token?

Tokenizing

- Issues cont.
  - Too much information lost?
    - Example:
      - “Bigcorp's 2007 bi-annual report showed profits rose 10%.” becomes
      - “Bigcorp 2007 annual report showed profits rose”
    - Small decisions in tokenizing can have major impact on effectiveness of some queries
  - Small words can be important in some queries, usually in combinations
    - E.g. ma, pm, el paso, master p, j lo, world war II

Tokenizing Problems

- Special characters are an important part of URLs, code in documents
- Capitalized words can have different meaning from lower case words
  - Bush, Apple
- Numbers can be important, including decimals
  - nokia 3250, top 10 courses, united 93, quicktime 6.5 pro, 92.3 the beat, 288358
- Periods can occur in numbers, abbreviations, URLs, ends of sentences, and other situations
  - I.B.M., Ph.D., cs.umass.edu, F.E.A.R.

Normalization

- Process of reducing the token to their "base" forms
  - Result is terms: a term is an entry in the IR system dictionary
- Need to "normalize" words in indexed text as well as query words in the same way
- Most common method to normalize is to implicitly create equivalence classes of terms by, e.g.,
  - deleting periods to form a term
    - ***U.S.A., USA***
  - deleting hyphens to form a term
    - ***anti-discriminatory, antidiscriminatory***

Normalization Techniques

- Stemming and Lemmatization uses morphology based rules for normalization
  - Morphology: “study of words, how they are formed, and their relationship to other words in the same language” [Wikipedia]
- Reduce terms to their “roots” before indexing
- Morphological variations of words include
  - *inflectional* (plurals, tenses)
    - Does not change core meaning of the word
    - e.g. see -> saw, goose -> geese
  - *derivational* (making verbs nouns etc.)
    - E.g. Happy -> unhappy, slow -> slowness (adjective to noun)
- Examples
  - car, cars, car’s, cars’ ⇒ car (stemming)
  - am, are, is ⇒be (lemmatization)

Stemming

- Stemmers attempt to reduce morphological variations of words to a common “stem”
  - usually involves affix chopping
- Two basic types
  - Dictionary-based: uses lists of related words
    - Performance depends on the dictionary used; can be slow
  - Algorithmic: uses program to determine related words
- Algorithmic stemmers
  - *suffix-s:* remove ‘s’ endings assuming plural
  - e.g., cats → cat, lakes → lake, wiis → wii
  - Many *false negatives*: supplies → supplie
  - Some *false positives*: ups → up

Lemmatization

- Use a vocabulary and morphological analysis of words to remove inflectional endings only and return the base or dictionary form of a word (lemma)
- “ saw ” -> “ see ” or “ saw ” depending on whether the token is used as a verb or a noun

Indexing

- Index: a data structure built from the text to speed up the searches
- In the context of an IR system that uses an index, the efficiency of the system can be measured by:
  - Indexing time: Time needed to build the index
  - Indexing space: Space used during the generation of the index
  - Index storage: Space required to store the index
  - Query latency: Time interval between the arrival of the query and the generation of the answer
  - Query throughput: Average number of queries processed per second

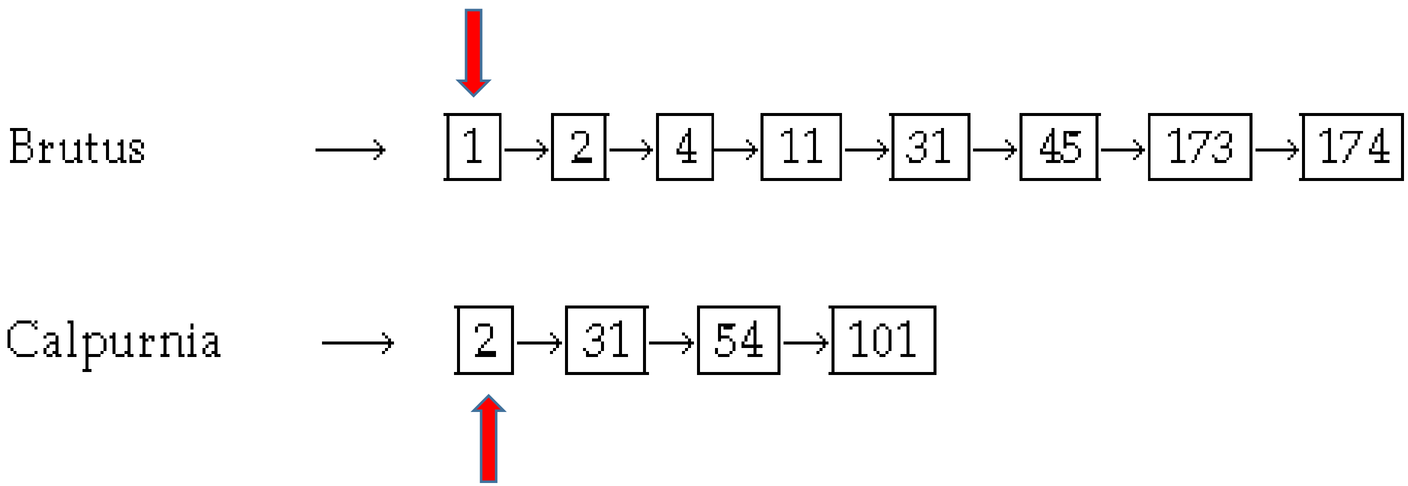
Indexer steps: Dictionary & Postings

- Multiple term entries in a single document are merged
- <u>Split into Dictionary and Postings</u>
- Doc. 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

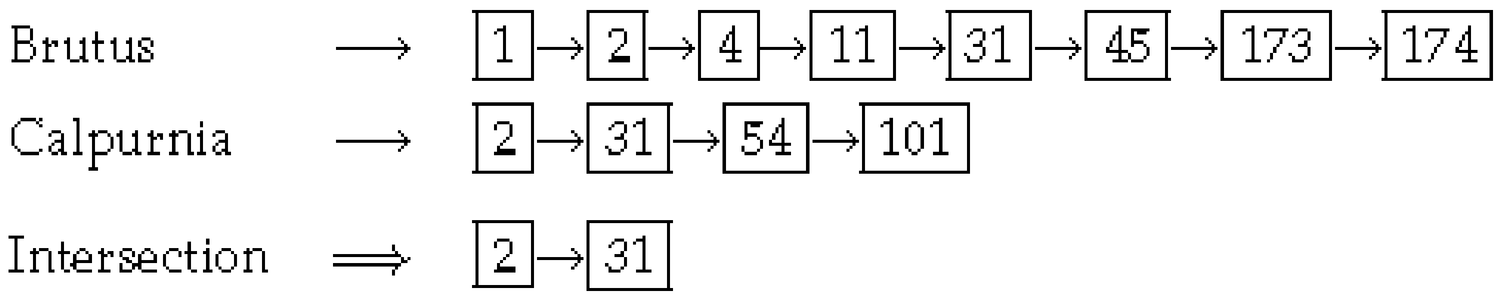
term	doc. freq.	→	postings lists
ambitious	1	→	2
be	1	→	2
brutus	2	→	1 → 2
capitol	1	→	1
caesar	2	→	1 → 2
did	1	→	1
enact	1	→	1
hath	1	→	2
i	1	→	1
i'	1	→	1
it	1	→	2
julius	1	→	1
killed	1	→	1
let	1	→	2
me	1	→	1
noble	1	→	2
so	1	→	2
the	2	→	1 → 2
told	1	→	2
you	1	→	2
was	2	→	1 → 2
with	1	→	2

Basic Merge of Posting List cont.



Brutus AND Calpurnia → {}

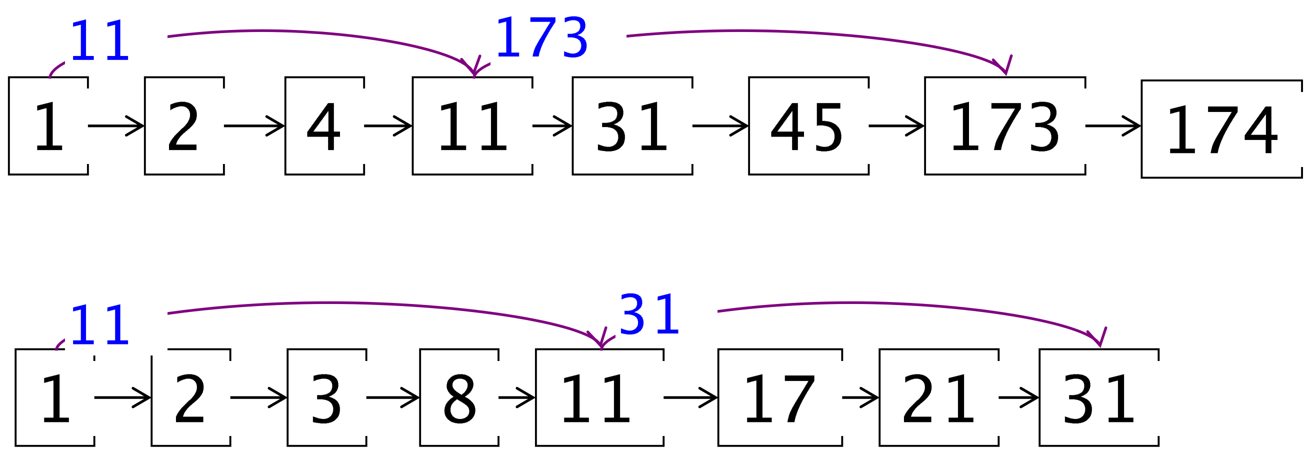
Basic Merge of Posting List cont.



Run time = ?

- If the list lengths are *m* and *n*, the merge takes O(*m*+*n*) operations.
- Can we do better?
  - Skip Pointers!

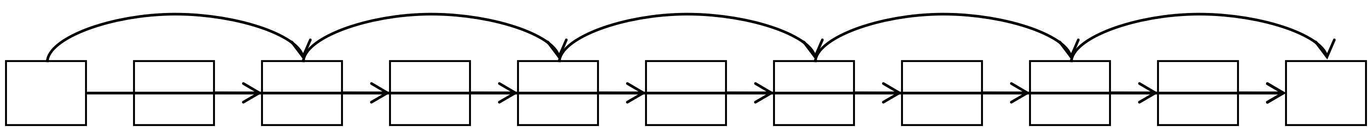
Skip Pointers



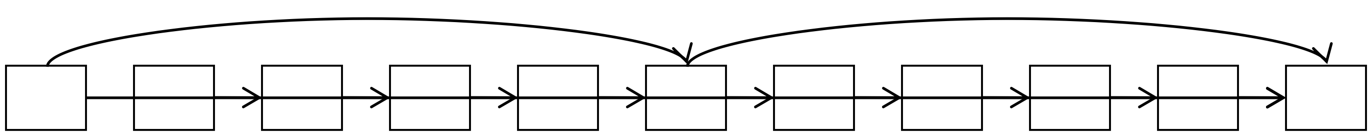
- Augment postings with skip pointers (at indexing time)
  - Why?
- <u>To skip postings that will not figure in the search results.</u>
  - How?
  - Where do we place skip pointers?

Tradeoffs with Skip Pointers

- Where to place the skip pointers:
- 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 or popularity of query terms
- Easy if the index is relatively static; harder if L keeps changing because of updates

Phrase Queries

- We want to answer a query such as [stanford university] – as a phrase.
  - False Positive: “The inventor Stanford Ovshinsky never went to university” should not be a match.
- Phrases are widely used
  - More precise than single words
    - e.g., documents containing “black sea” vs. two words “black” and “sea”
  - Less ambiguous
    - e.g., “big apple” vs. “apple”
  - Changing the order of the words will create a completely different phrase.
    - E.g. “The fox is brown” is different from “Is the fox brown”
- Significant part of web queries are phrase queries, either explicitly entered (within quotes) or interpreted as such

Phrase Indexes

- Idea: phrase is sequence of *n* words, otherwise called word *n-grams*
  - *bigram*: 2 word sequence, *trigram*: 3 word sequence, *unigram*: single words
  - N-grams can be generated at character level (useful in wildcard index matching, later)
- Indexes generated with word n-grams are called "phrase index"
- Frequent n-grams are more likely to be meaningful phrases
- Possible index all n-grams up to specified length
  - Uses a lot of storage
  - e.g., document containing 1,000 words would contain 3,990 instances of word n-grams of length $2 \le n \le 5$
- Much faster than POS tagging

Positional Indexing

- Positional indexes are a more efficient alternative to phrase indexes
- Normally, in postings lists each posting is just a docID
- Each posting in a positional index contains:
  - docID
  - list of positions in the document where the term is found

Positional Indexes: Proximity Search

- We can 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 match
  - "*Employment agencies that have learned to adapt now place healthcare workers*" is not a match

Dictionary Data Structure

- Two questions when designing the dictionary
  - How do we store a dictionary in memory efficiently?
  - How do we quickly look up elements at query time?
- Potential Data structures to store indexes
  - Hash Table
  - Binary Tree
  - B-tree
- Criteria for when to use hashes vs. trees:
  - Is there a fixed number of terms or will it keep growing?
  - What are the relative frequencies with which various keys will be accessed?
  - How many terms are we likely to have?

Tolerant Retrieval

- User hate typing
  - Queries include wildcards *
  - Spelling mistakes in the query
- IR systems should work with "imperfect" queries
  - Tolerant retrieval

Wildcard queries: *

- ***mon*:*** find all docs containing any word beginning with "mon". How?
  - Easy with binary tree (or B-tree) dictionary: retrieve all words in range: *mon* **≤ w <** *moo*
- ***mon:*** find words ending in "mon" How?
  - Harder
  - Maintain an additional tree for terms *backwards*
  - Can retrieve all words in range: ***nom ≤ w < non***

Permuterm index

- Basic idea: Rotate all wildcard query term, so that the * occurs at the end.
  - But we should have each of these rotations in the dictionary already
  - Example: for the term ***hello***, index under:
    - ***hello$, ello$h, llohe, lohel, o$hell, $hello***
    - where $ is a special symbol, indicating end of a term

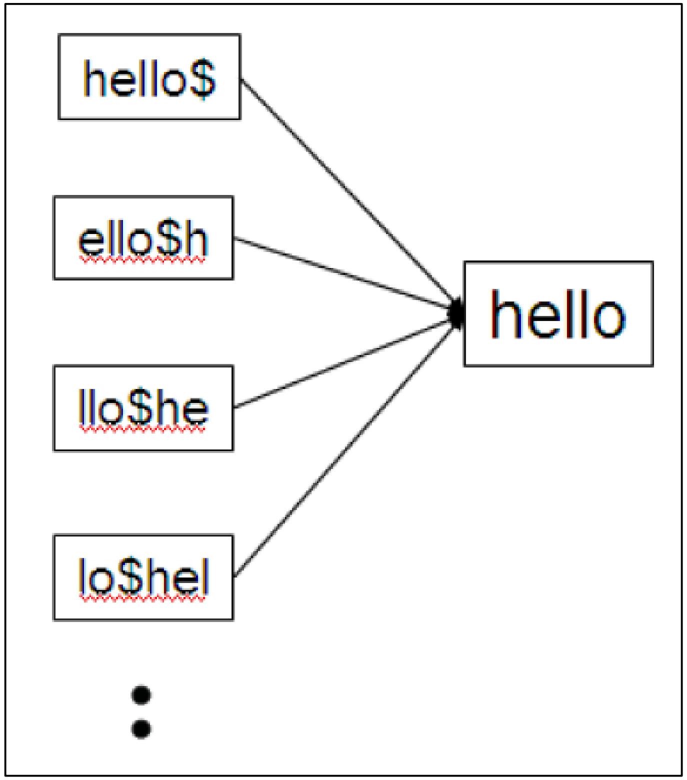


Fig. Permuterm Index

Bigram (k-gram) indexes

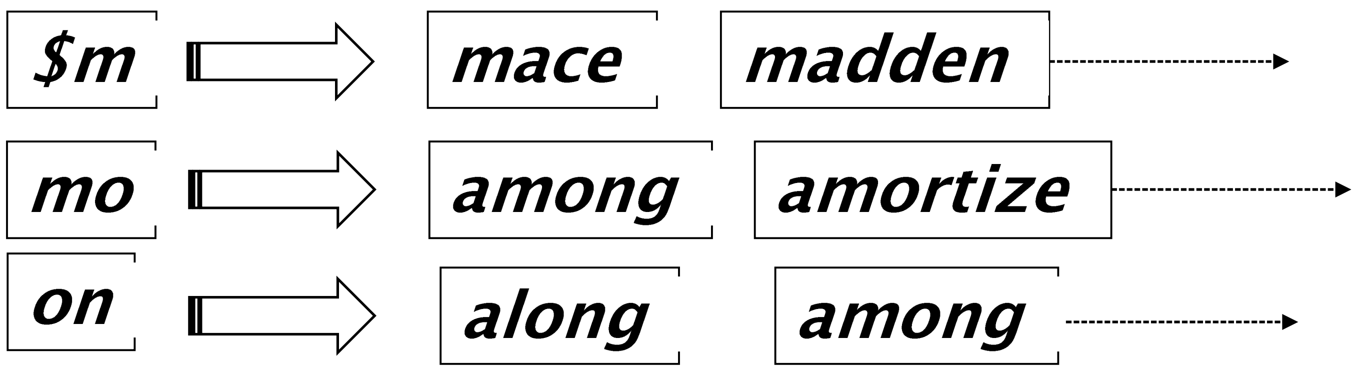
- Enumerate all *k*-grams (sequence of *k* chars) occurring in any term
  - *e.g.,* from text "***April is the cruelest month***" we get the 2-grams (*bigrams*)

•a,ap,pr,ri,il,l,i,is,s,t,th,he,e,$c,cr,ru,
•ue,el,le,es,st,t$, m,mo,on,nt,h

  - $ is a special word boundary symbol, as used before in permuterm indexing
- Maintain a <u>*second*</u> inverted index <u>*from bigrams to dictionary terms*</u> that match each bigram.

Bigram index example

- The *k*-gram index finds *terms* based on a query ("moon") consisting of *k*-grams (here *k*=2).



Spell correction

- Two principal uses
  - Correcting document(s) being indexed
  - Correcting user queries to retrieve "right" answers
- Two main types of spell correction:
  - Isolated word
    - Check each word on its own for misspelling
    - Will not catch typos resulting in correctly spelled words
    - e.g., ***from*** $\rightarrow$ ***form***
  - Context-sensitive
    - Look at surrounding words,
    - e.g., ***I flew <u>form</u> Heathrow to Narita.***

Edit distance

- Given two strings *S1* and *S2*, the minimum number of operations to convert one to the other
- Operations are typically character-level
  - Insert, Delete, Replace
- E.g., the edit distance from ***dof*** to ***dog*** is 1
  - From ***cat*** to ***act*** is 2 (Just 1 with transpose.)
  - from ***cat*** to ***dog*** is 3.

Levenshtein distance: Example

		f	a	s	t
	0	**1** **1**	**2** **2**	**3** **3**	**4** **4**
c	**1** **1**	1 2 2 1	**2** 3 **2** **2**	**3** 4 **3** **3**	**4** 5 **4** **4**
a	**2** **2**	**2** **2** 3 **2**	1 3 3 1	3 4 2 2	4 5 **3** **3**
t	**3** **3**	**3** **3** 4 **3**	3 **2** 4 **2**	2 3 3 2	2 4 3 2
s	**4** **4**	**4** **4** 5 **4**	4 **3** 5 **3**	**2** 3 4 **2**	3 3 3 3

Each cell of Levenshtein matrix

cost of getting here from my upper left neighbor (copy or replace)	cost of getting here from my upper neighbor (delete)
cost of getting here from my left neighbor (insert)	the minimum of the three possible “movements”; the cheapest way of getting here

		f	a	s	t
	0	1 1	2 2	3 3	4 4
c	1 / 1	1 2 / 2 1	2 3 / 2 2	3 4 / 3 3	4 5 / 4 4
a	2 / 2	2 2 / 3 2	1 3 / 3 1	3 4 / 2 2	4 5 / 3 3
t	3 / 3	3 3 / 4 3	3 2 / 4 2	2 3 / 3 2	2 4 / 3 2
s	4 / 4	4 4 / 5 4	4 3 / 5 3	2 3 / 4 2	3 3 / 3 3

n-gram overlap

- Enumerate all the *n*-grams in the query string as well as in the lexicon
- Use the *n*-gram index (recall wild-card search) to retrieve all lexicon terms matching any of the query *n*-grams
- Threshold by number of matching *n*-grams
  - Variants – weight by keyboard layout, etc.
- Suppose the text is ***november***
  - Trigrams are *nov, ove, vem, emb, mbe, ber*.
- The misspelled query is ***nawember***
  - Trigrams are *naw, awe, wem, emb, mbe, ber*.
- So 3 trigrams overlap (of 6 in each term)
- How can we turn this into a normalized measure of overlap?
  - Jaccard coefficient

Index construction: Naïve Algorithm

- Parsing: one document at a time.
  - Final posting lists for any term are incomplete until the end.
- Potential problem: Can we keep all postings in memory and then do the sort in-memory at the end?
  - No, not for large collections
- Solution: ?
  - intermediate results on disk.
- Problem: Disk I/O is slow!

BSBI cont.

BSBIndexConstruction()
$n \leftarrow 0$
while (all documents have not been processed)
do $n \leftarrow n + 1$
    block $\leftarrow$ ParseNextBlock()
    BSBI-Invert(*block*)
    WriteBlockToDisk(*block*, f_n)
MergeBlocks($f_1, \ldots, f_n; f_{\text{merged}}$)

- parses documents into termID–docID pairs
- termID – a unique ID for the terms
- Need mapping between term and termID

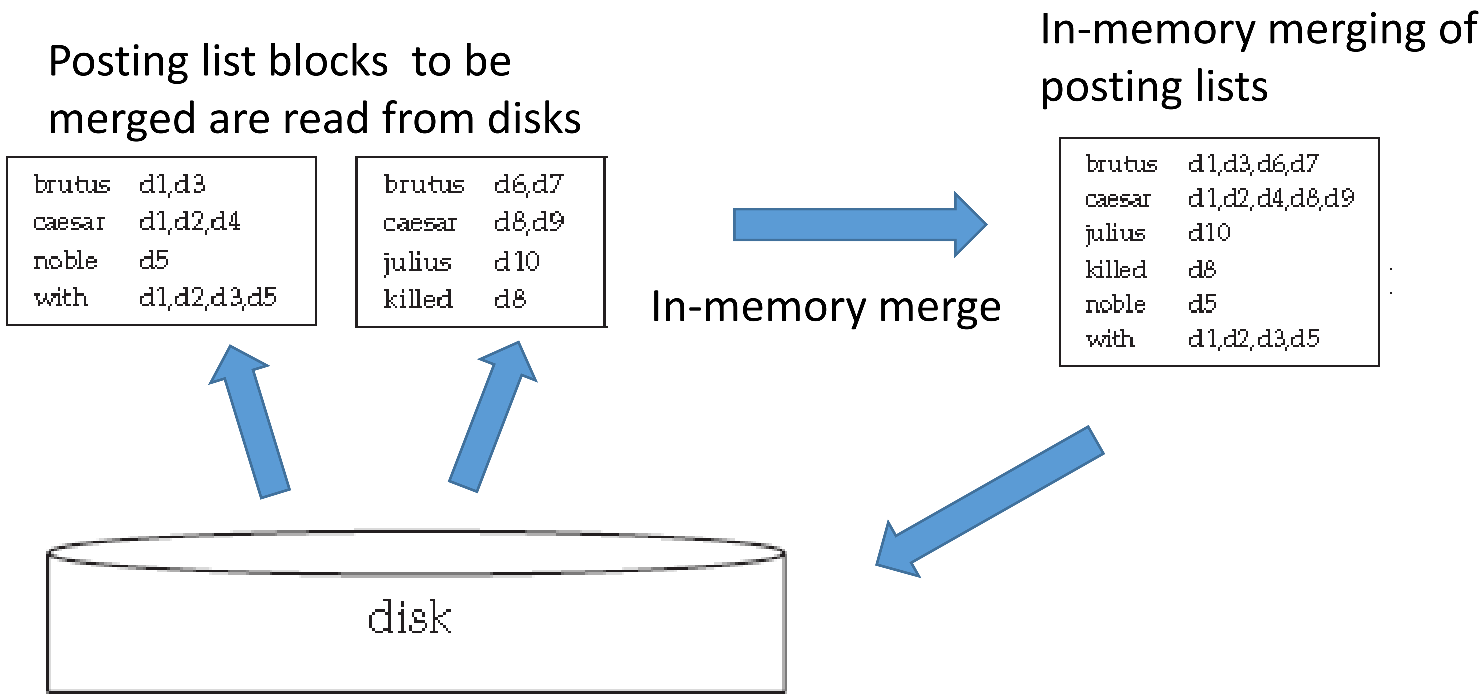
- Sort the termID–docID pairs
- Collect termID–docID pairs with the same termID
- Generate postings list

- Disk I/O

- Next slide

Blocked Sort-Based Indexing (BSBI)

Merge Phase



Single-pass in-memory indexing (SPIMI)

- Key idea 1: Generate separate (incomplete dictionaries) for each block – no need to maintain term-termID mapping across blocks.
- Key idea 2: Don’t sort. Accumulate postings in postings lists as they occur.
- With these two ideas we can generate a complete inverted index for each block.
- These separate indexes can then be merged into one big index.

SPIMI cont.

```
SPIMI-Invert(token stream)
output file ← NewFile()
dictionary ← NewHash()
while (free memory available)
do token ← next(token_stream)
     if term(token) ∉ dictionary
          then postings list ← AddToDictionary(dictionary ,term(token))
          else postings list ← GetPostingsList(dictionary ,term(token))
     if full (postings list)
          then postings list ← DoublePostingsList(dictionary ,term(token))
     AddToPostingsList(postings list,docID(token))
sorted terms ← SortTerms(dictionary )
WriteBlockToDisk(sorted terms,dictionary ,output file)
return output file
```

- Merging of blocks is analogous to BSBI.

Why compression for inverted indexes?

- Compress Dictionary
  - If possible: Make it small enough to keep in main memory
    - Speeds up retrieval
  - With large corpus: Make it so small that you can keep some postings lists in main memory too
- Compress Postings file(s)
  - Reduce disk space needed
  - Decrease time needed to read postings lists from disk
- Compression lets you keep more in memory
  - Large search engines keep a significant part of the postings in memory.
- Discuss various IR-specific compression schemes

44

Vocabulary vs. collection size

- How big is the term vocabulary?
- That is, how many distinct terms are there?
- In practice, the vocabulary will keep growing with the collection size
- Heaps' law: $M = kT^b$
  - M is the size of the vocabulary, T is the number of tokens in the collection
  - Typical values: $30 \le k \le 100$ and $b \approx 0.5$

45

Zipf's law

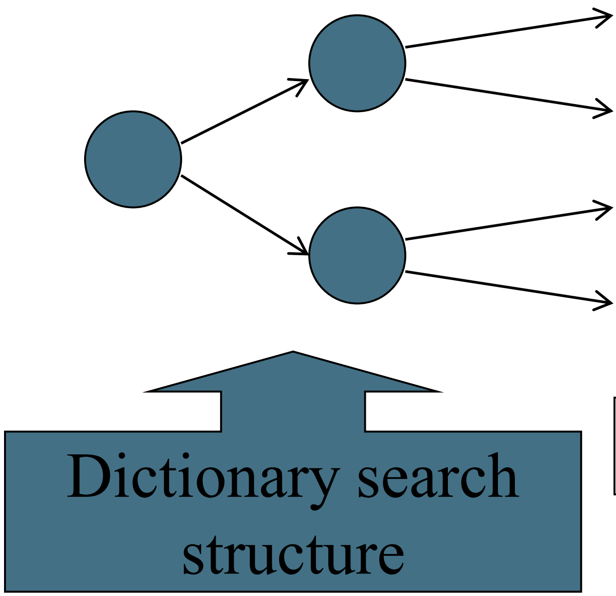
- Heaps' law gives the vocabulary size in collections.
- How about the relative frequencies of terms?
  - In natural language, there are a few very frequent terms and very many very rare terms.
- Zipf's law: The i^{th} most frequent term has frequency proportional to $\frac{1}{i}$.
  - $cf_i \propto \frac{1}{i} = \frac{k}{i}$ where k is a normalizing constant
    - cf_i is <u>collection frequency</u>: the number of occurrences of the term t_i in the collection.

Why compress the dictionary?

- Search begins with the dictionary
  - Keep it in memory
- Other constraints:
  - Memory footprint competition with other applications
- Even if the dictionary isn't in memory, we want it to be small for a fast search startup time
- We will consider compressing the space for both dictionary and postings
  - Consider only non-positional indexes

Dictionary storage - first cut

- Array of fixed-width entries
- ~400,000 terms; 28 bytes/term = 11.2 MB.



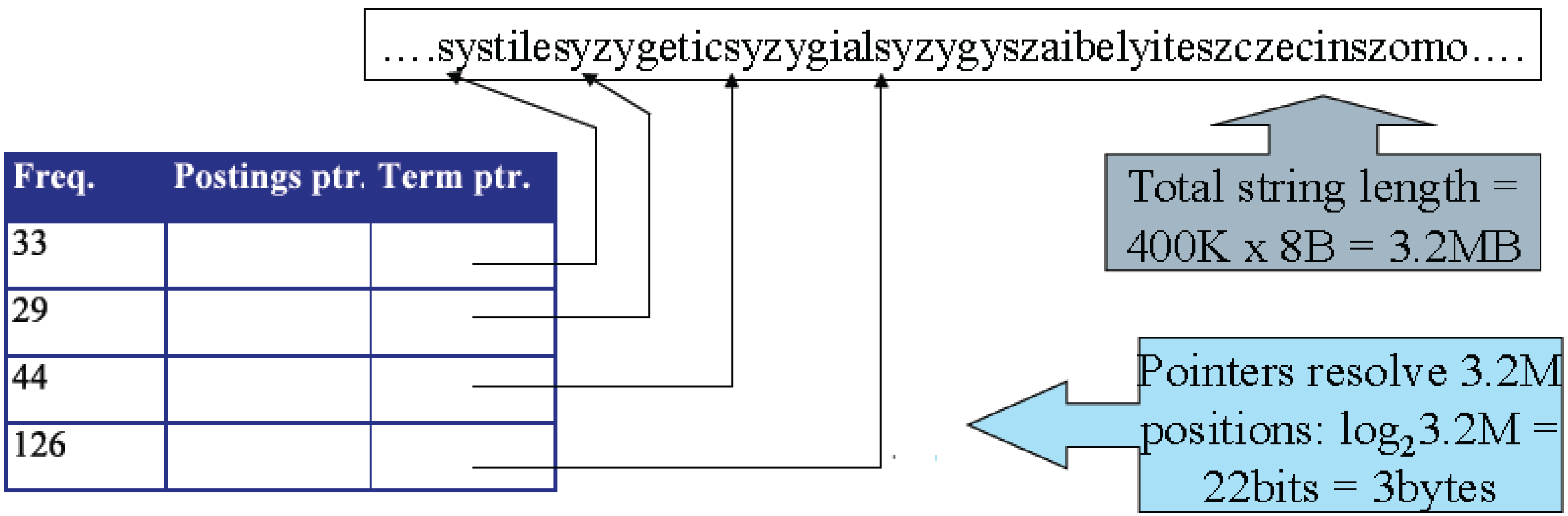
term	document frequency	pointer to postings list
a	656,265	$\longrightarrow$
aachen	65	$\longrightarrow$
...	...	...
zulu	221	$\longrightarrow$

20 bytes

4 bytes each

Compressing term list: Dictionary-as-a-String

- How do store dictionary with each term ~ 8 Bytes?

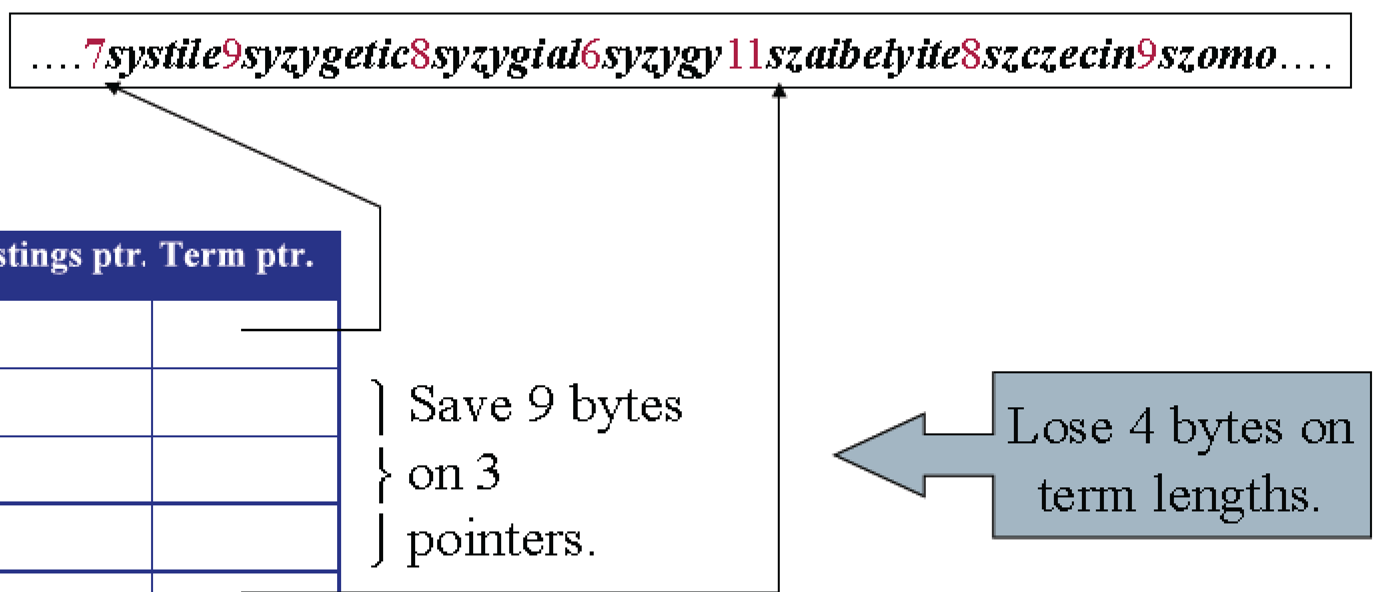


- Store dictionary as a (long) string of characters:
- Pointer to next word shows end of current word
- Save up to 60% of dictionary space

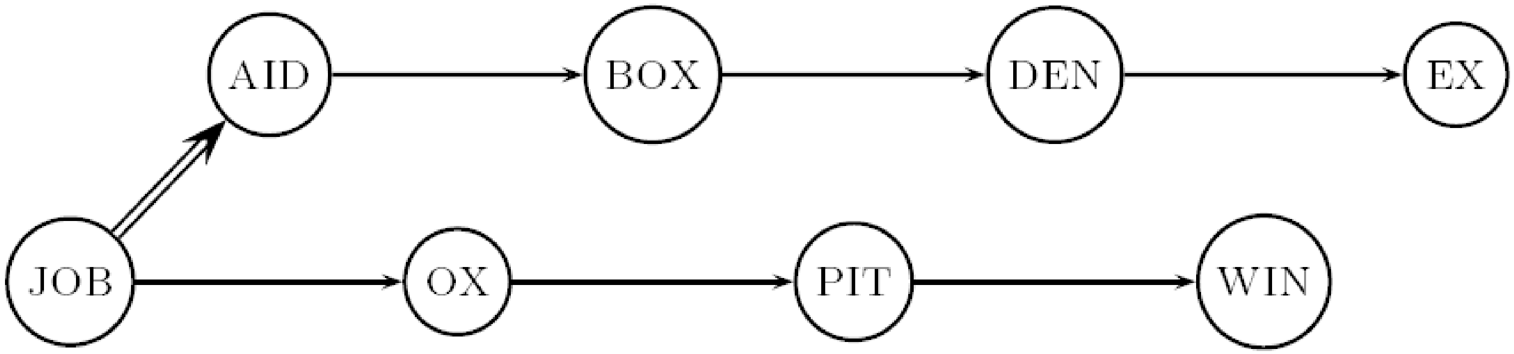
Blocking

- Store pointers to every *k*th term string.
- Example below: *k*=4.
- Need to store term lengths (1 extra byte)

Freq.	Postings ptr.	Term ptr.
33		
29		
44		
126		
7		



Dictionary Search with Blocking



- Binary search down to 4-term block;
  - Then linear search through terms in block.
  - Blocks of 4 (binary tree), avg. = (1+2·2+2·3+2·4+5)/8 = 3 compares

Postings compression

- The postings file storage requirement much larger than the dictionary, factor of at least 10.
- Compression: store each posting compactly.
  - A posting for our purposes is a docID.
- For Reuters (800K documents), we would use 32 bits per docID when using 4-byte integers.
- Alternatively, we can use $\log_2 800{,}000 \approx 20$ bits per docID.
- Our goal: use far fewer than 20 bits per docID.

Variable Byte (VB) codes

- For a gap value *G,* we want to use close to the fewest bytes needed to hold $\log_2 G$ bits
- Begin with one byte to store *G* and dedicate 1 bit in it to be a <u>continuation</u> bit *c*
- If $G \leq 127$, binary-encode it in the 7 available bits and set $c = 1$
- Else encode *G*'s lower-order 7 bits and then use additional bytes to encode the higher order bits using the same algorithm
- At the end set the continuation bit of the last byte to 1 ($c = 1$) – and for the other bytes $c = 0$.