ADFOCS 2004

Prabhakar Raghavan

Lecture 1

Plan: Basic information retrieval

- Lecture 1: ~120 minutes
  - Index structures
- Lecture 2: 90 minutes
  - Index compression and construction
- Lecture 3: ~90 minutes
  - Scoring and evaluation

Query

- Which plays of Shakespeare contain the words ***Brutus*** *AND* ***Caesar*** but *NOT* ***Calpurnia***?
- Could grep all of Shakespeare's plays for ***Brutus*** and ***Caesar,*** then strip out lines containing ***Calpurnia***?
  - Slow (for large corpora)
  - *NOT* ***Calpurnia*** is non-trivial
  - Other operations (e.g., find the phrase ***Romans and countrymen***) not feasible

Term-document incidence

	Antony and Cleopatra	Julius Caesar	The Tempest	Hamlet	Othello	Macbeth
Antony	1	1	0	0	0	1
Brutus	1	1	0	1	0	0
Caesar	1	1	0	1	1	1
Calpurnia	0	1	0	0	0	0
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0

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.

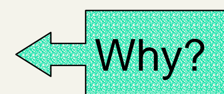
- Hamlet, Act III, Scene ii
- *Lord Polonius:* I did enact Julius ***Caesar*** I was killed i' the
- Capitol; ***Brutus*** killed me.

Bigger corpora

- Consider *n* = 1M documents, each with about 1K terms.
- Avg 6 bytes/term incl spaces/punctuation
  - 6GB of data in the documents.
- Say there are *m* = 500K *distinct* terms among these.

Can't build the matrix

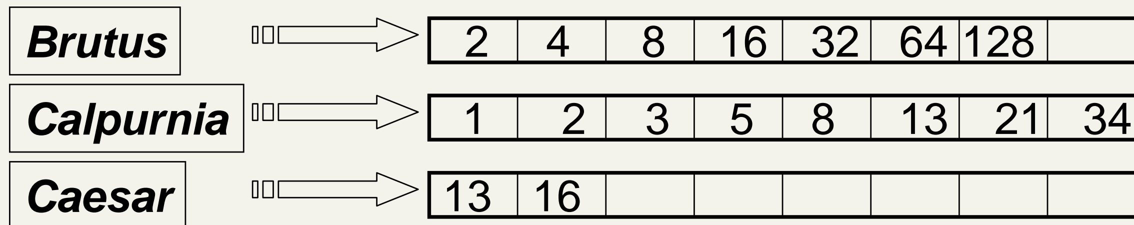
- 500K x 1M matrix has half-a-trillion 0's and 1's.
- But it has no more than one billion 1's.
  - matrix is extremely sparse.



- What's a better representation?
  - We only record the 1 positions.

Inverted index

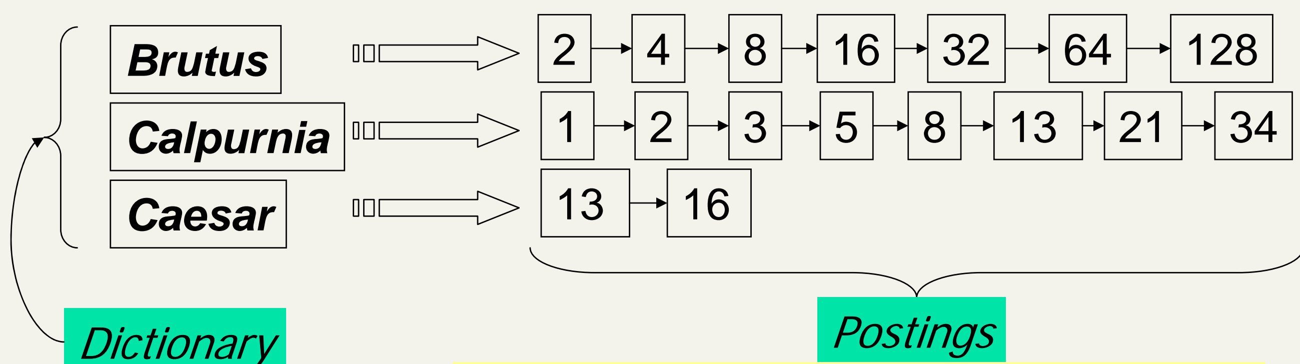
- For each term *T*, must store a list of all documents that contain *T*.
- Do we use an array or a list for this?



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

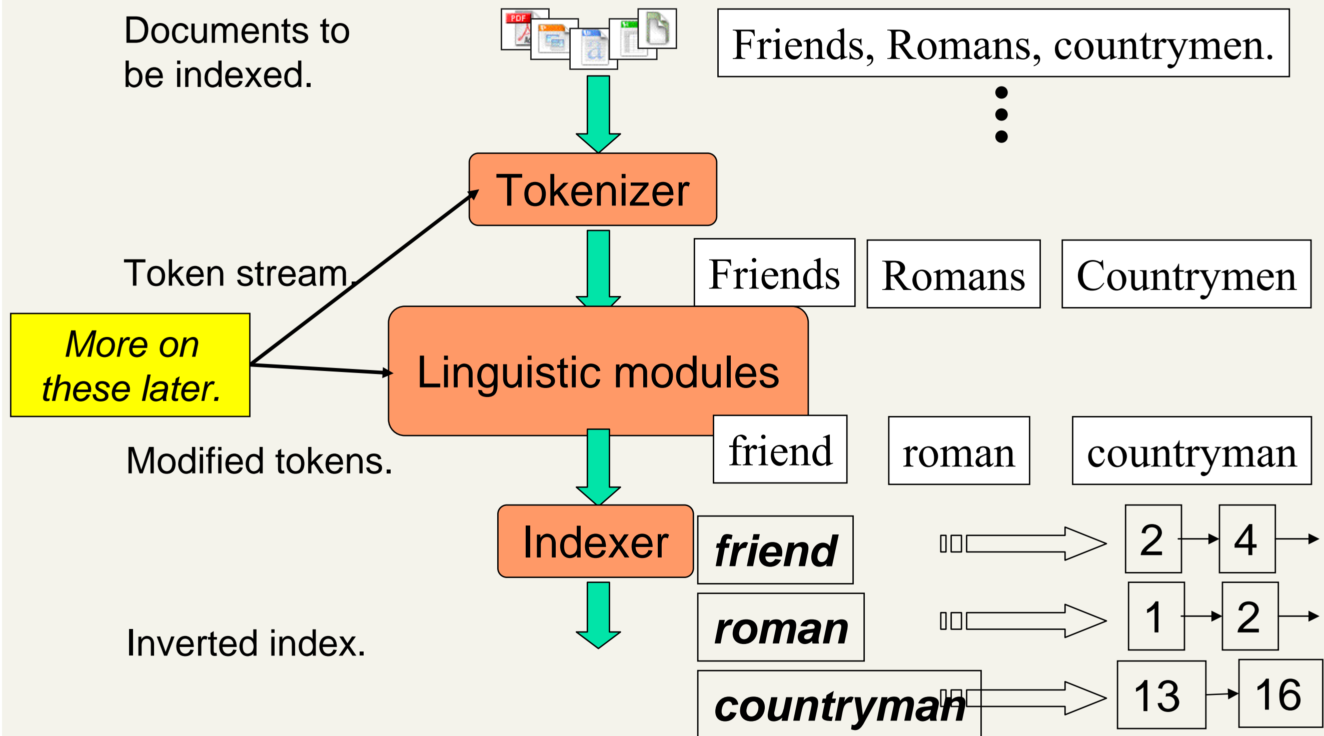
Inverted index

- Linked lists generally preferred to arrays
  - Dynamic space allocation
  - Insertion of terms into documents easy
  - Space overhead of pointers



Sorted by docID (more later on why).

Inverted index construction



Indexer steps

- 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

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

- Sort by terms.

Core indexing step.

Term	Doc #
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

→

Term	Doc #
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

- Multiple term entries in a single document are merged.
- Frequency information is added.

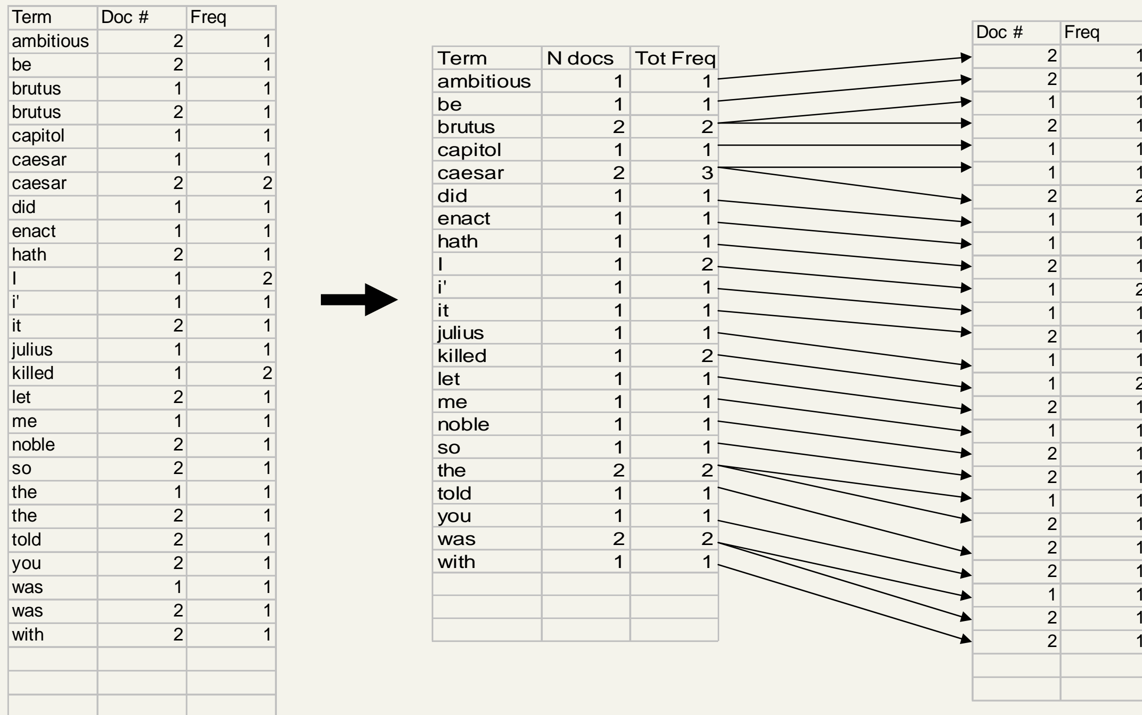
Why frequency? Will discuss later.

Term	Doc #
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
ambitious	2	1
be	2	1
brutus	1	1
brutus	2	1
capitol	1	1
caesar	1	1
caesar	2	2
did	1	1
enact	1	1
hath	2	1
I	1	2
i'	1	1
it	2	1
julius	1	1
killed	1	2
let	2	1
me	1	1
noble	2	1
so	2	1
the	1	1
the	2	1
told	2	1
you	2	1
was	1	1
was	2	1
with	2	1

- The result is split into a *Dictionary* file and a *Postings* file.

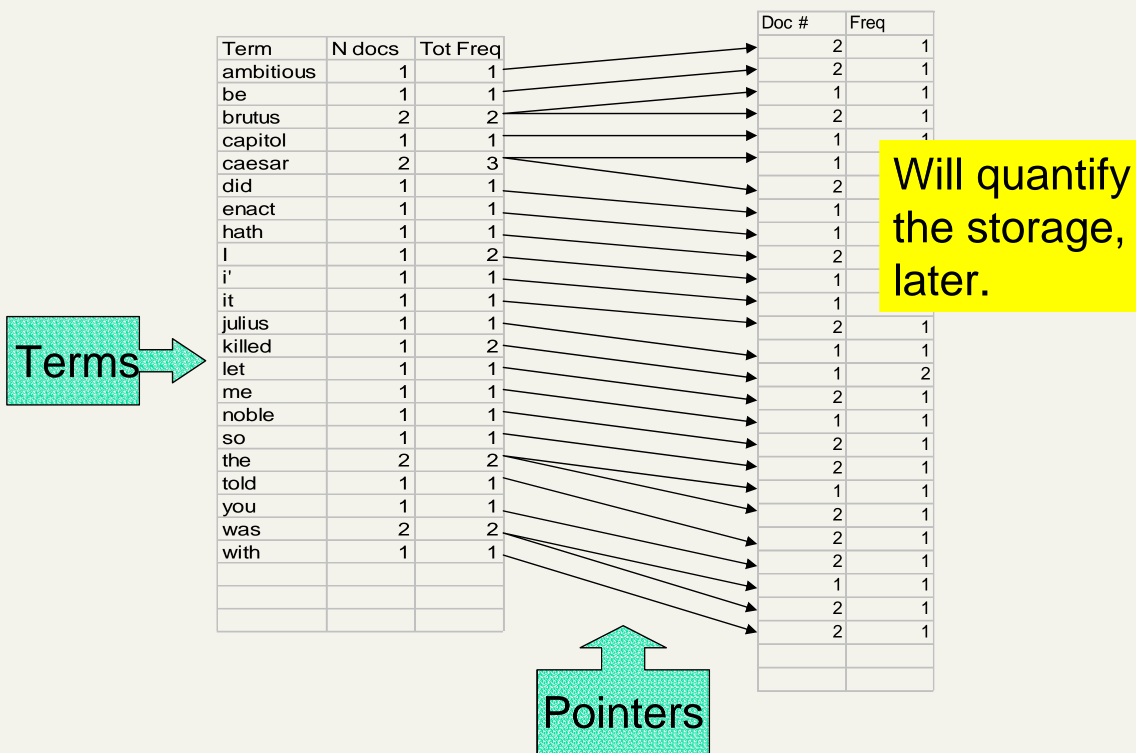


Term	Doc #	Freq
ambitious	2	1
be	2	1
brutus	1	1
brutus	2	1
capitol	1	1
caesar	1	1
caesar	2	2
did	1	1
enact	1	1
hath	2	1
I	1	2
i'	1	1
it	2	1
julius	1	1
killed	1	2
let	2	1
me	1	1
noble	2	1
so	2	1
the	1	1
the	2	1
told	2	1
you	2	1
was	1	1
was	2	1
with	2	1

Term	N docs	Tot Freq
ambitious	1	1
be	1	1
brutus	2	2
capitol	1	1
caesar	2	3
did	1	1
enact	1	1
hath	1	1
I	1	2
i'	1	1
it	1	1
julius	1	1
killed	1	2
let	1	1
me	1	1
noble	1	1
so	1	1
the	2	2
told	1	1
you	1	1
was	2	2
with	1	1

Doc #	Freq
2	1
2	1
1	1
2	1
1	1
1	1
2	2
1	1
1	1
2	1
1	2
1	1
2	1
1	1
1	2
2	1
1	1
2	1
2	1
1	1
2	1
2	1
2	1
1	1
2	1
2	1

- Where do we pay in storage?



Term	N docs	Tot Freq
ambitious	1	1
be	1	1
brutus	2	2
capitol	1	1
caesar	2	3
did	1	1
enact	1	1
hath	1	1
I	1	2
i'	1	1
it	1	1
julius	1	1
killed	1	2
let	1	1
me	1	1
noble	1	1
so	1	1
the	2	2
told	1	1
you	1	1
was	2	2
with	1	1

Doc #	Freq
2	1
2	1
1	1
2	1
1	1
1	
2	
1	
1	
2	
1	
1	
2	1
1	1
1	2
2	1
1	1
2	1
2	1
1	1
2	1
2	1
2	1
1	1
2	1
2	1

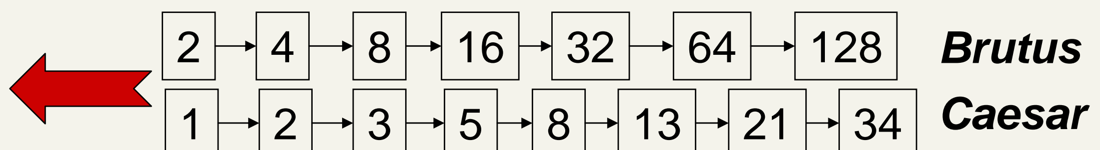
The index we just built

- How do we process a query?
  - What kinds of queries can we process?
- Which terms in a doc do we index?
  - All words or only "important" ones?
- Stopword list: terms that are so common that they're ignored for indexing.
  - *e.g.*, ***the, a, an, of, to*** …
  - language-specific.

Query processing

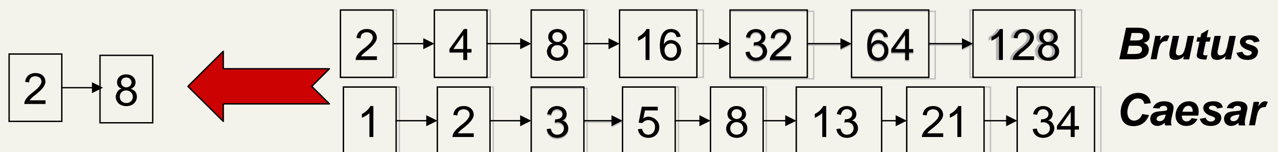
- Consider processing the query:

  Brutus *AND* ***Caesar***
  - Locate ***Brutus*** in the Dictionary;
    - Retrieve its postings.
  - Locate *Caesar* in the Dictionary;
    - Retrieve its postings.
  - "Merge" the two postings:



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.

Boolean queries: Exact match

- Queries using *AND, OR* and *NOT* together with query terms
  - Views each document as a set of words
  - Is precise: document matches condition or not.
- Primary commercial retrieval tool for 3 decades.
- Professional searchers (e.g., Lawyers) still like Boolean queries:
  - You know exactly what you're getting.

Example: WestLaw http://www.westlaw.com/

- Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992)
- About 7 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
- Long, precise queries; proximity operators; incrementally developed; not like web search

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)?

Merging

What about an arbitrary Boolean formula?

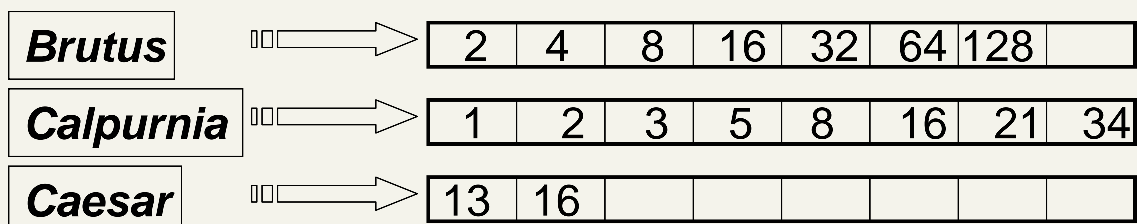
***(Brutus** OR **Caesar)** AND NOT*

(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 *t* terms.
- For each of the *t* terms, get its postings, then *AND* together.

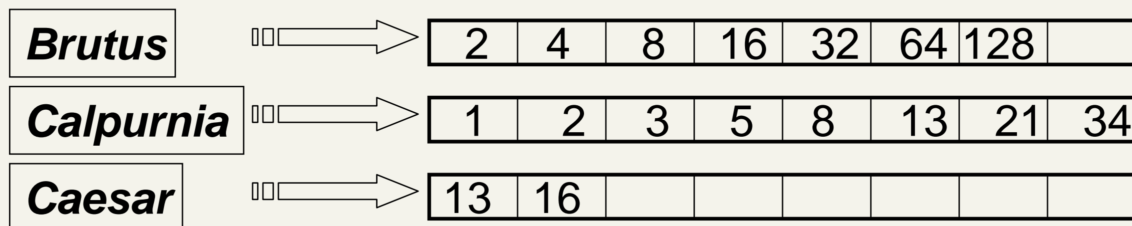


Query: ***Brutus** AND **Calpurnia** AND **Caesar***

Query optimization example

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

This is why we kept
freq in dictionary



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

More general optimization

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

Exercise

- Recommend a query processing order for

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

Term	Freq
eyes	213312
kaleidoscope	87009
marmalade	107913
skies	271658
tangerine	46653
trees	316812

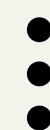
Query processing exercises

- If the query is ***friends*** *AND* ***romans*** *AND (NOT* ***countrymen****)*, how could we use the freq of ***countrymen***?
- Exercise: Extend the merge to an arbitrary Boolean query. Can we always guarantee execution in time linear in the total postings size?
- Hint: Begin with the case of a Boolean *formula* query: the each query term appears only once in the query.

Online Exercise

- Try the search feature at http://www.rhymezone.com/shakespeare/
- Write down five search features you think it could do better

Recall basic indexing pipeline



Tokenization

Tokenization

- Input: "***Friends, Romans and Countrymen***"
- Output: Tokens
  - ***Friends***
  - ***Romans***
  - ***Countrymen***
- Each such token is now a candidate for an index entry, after further processing
  - Described below
- But what are valid tokens to emit?

Parsing a document

- What format is it in?
  - pdf/word/excel/html?
- What language is it in?
- What character set is in use?

Each of these is a classification problem, which we will study later in the course.

But there are complications …

Format/language stripping

- Documents being indexed can include docs from many different languages
  - A single index may have to contain terms of several languages.
- Sometimes a document or its components can contain multiple languages/formats
  - French email with a Portuguese pdf attachment.
- What is a unit document?
  - An email?
  - With attachments?
  - An email with a zip containing documents?

Tokenization

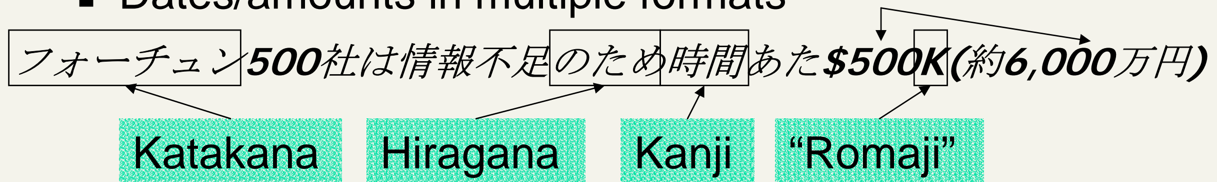
- Issues in tokenization:
  - ***Finland's capital*** → ***Finland? Finlands? Finland's***?
  - ***Hewlett-Packard*** → ***Hewlett*** and ***Packard*** as two tokens?
  - ***San Francisco***: one token or two? How do you decide it is one token?

Language issues

- Accents: ***résumé*** vs. ***resume***.
- ***L'ensemble*** → one token or two?
  - ***L*** ? ***L'*** ? ***Le*** ?
- How are your users like to write their queries for these words?

Tokenization: language issues

- Chinese and Japanese have no spaces between words:
  - Not always guaranteed a unique tokenization
- Further complicated in Japanese, with multiple alphabets intermingled
  - Dates/amounts in multiple formats



End-user can express query entirely in (say) Hiragana!

Normalization

- In "right-to-left languages" like Hebrew and Arabic: you can have "left-to-right" text interspersed (e.g., for dollar amounts).
- Need to "normalize" indexed text as well as query terms into the same form

  7月30日 vs. 7/30

- Character-level alphabet detection and conversion
  - Tokenization not separable from this.
  - Sometimes ambiguous:

    Morgen will ich in MIT …

    Is this German "mit"?

Punctuation

- ***Ne'er***: use language-specific, handcrafted "locale" to normalize.
  - Which language?
  - Most common: detect/apply language at a pre-determined granularity: doc/paragraph.
- ***State-of-the-art***: break up hyphenated sequence. Phrase index?
- ***U.S.A.*** vs. ***USA*** - use locale.
- ***a.out***

Numbers

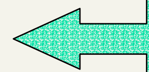
- ***3/12/91***
- ***Mar. 12, 1991***
- ***55 B.C.***
- ***B-52***
- ***My PGP key is 324a3df234cb23e***
- ***100.2.86.144***
  - Generally, don't index as text.
  - Will often index "meta-data" separately
    - Creation date, format, etc.

Case folding

- Reduce all letters to lower case
  - exception: upper case (in mid-sentence?)
    - *e.g.*, ***General Motors***
    - ***Fed*** vs. ***fed***
    - ***SAIL*** vs***. sail***

Spell correction

- Expand to terms within (say) edit distance 2
  - Edit∈{Insert/Delete/Replace}
  - Expand at query time from query
  - *e.g.*, ***Alanis Morisette***
- Spell correction is expensive and slows the query (upto a factor of 100)
  - Invoke only when index returns (near-)zero matches.
  - What if docs contain mis-spellings?



Why not at index time?

Thesauri and soundex

- Handle synonyms and homonyms
  - Hand-constructed equivalence classes
    - e.g., ***car*** = ***automobile***
    - ***your*** [symbol] ***you’re***
- Index such equivalences
  - When the document contains ***automobile***, index it under ***car*** as well (usually, also vice-versa)
- Or expand query?
  - When the query contains ***automobile***, look under ***car*** as well
- More on this later ...

Soundex

- Class of heuristics to expand a query into phonetic equivalents
  - Language specific – mainly for names
  - E.g., ***chebyshev*** → ***tchebycheff***

Soundex – typical algorithm

- Turn every token to be indexed into a 4-character reduced form
- Do the same with query terms
- Build and search an index on the reduced forms
  - (when the query calls for a soundex match)

- http://www.creativyst.com/Doc/Articles/SoundEx1/SoundEx1.htm#Top

Soundex – typical algorithm

1. Retain the first letter of the word.
2. Change all occurrences of the following letters to '0' (zero):
   'A', E', 'I', 'O', 'U', 'H', 'W', 'Y'.
3. Change letters to digits as follows:

- B, F, P, V → 1
- C, G, J, K, Q, S, X, Z → 2
- D,T → 3
- L → 4
- M, N → 5
- R → 6

Soundex continued

4. Remove all pairs of consecutive digits.
5. Remove all zeros from the resulting string.
6. Pad the resulting string with trailing zeros and return the first four positions, which will be of the form <uppercase letter> <digit> <digit> <digit>.

E.g., Herman becomes H655.

Query expansion

- Usually do query expansion rather than index expansion
  - No index blowup
  - Query processing slowed down
    - Docs frequently contain equivalences
  - May retrieve more junk
    - ***puma*** → ***jaguar*** retrieves documents on cars instead of on sneakers.

Language detection

- Many of the components described require language detection
  - For docs/paragraphs at indexing time
  - For query terms at query time – much harder
- For docs/paragraphs, generally have enough text to apply machine learning methods
- For queries, generally lack sufficient text
  - Augment with other cues, such as client properties/specification from application
  - Domain of query origination, etc.

What queries can we process?

- We have
  - Basic inverted index with skip pointers
  - Wild-card index
  - Spell-correction
- Queries such as

an*er* *AND* ***(moriset /3 toronto) OR SOUNDEX(chaikofski)***

Aside – results caching

- If 25% of your users are searching for

  britney *AND* ***spears***

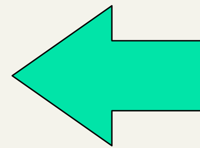
  then you probably *do* need spelling correction, but you *don't* need to keep on intersecting those two postings lists
- Web query distribution is extremely skewed, and you can usefully cache results for common queries – more later.

Lemmatization

- Reduce inflectional/variant forms to base form
- E.g.,
  - *am, are, is* → *be*
  - *car, cars, car's, cars'* → *car*
- *the boy's cars are different colors* → *the boy car be different color*

Dictionary entries – first cut

ensemble.french
時間.japanese
MIT.english
mit.german
guaranteed.english
entries.english
sometimes.english
tokenization.english



These may be grouped by language. More on this in query processing.

Stemming

- Reduce terms to their "roots" before indexing
  - 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 compres and compres are both accept as equival to compres.

Porter's algorithm

- Commonest algorithm for stemming English
- 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

- *sses* $\rightarrow$ *ss*
- *ies* $\rightarrow$ *i*
- *ational* $\rightarrow$ *ate*
- *tional* $\rightarrow$ *tion*

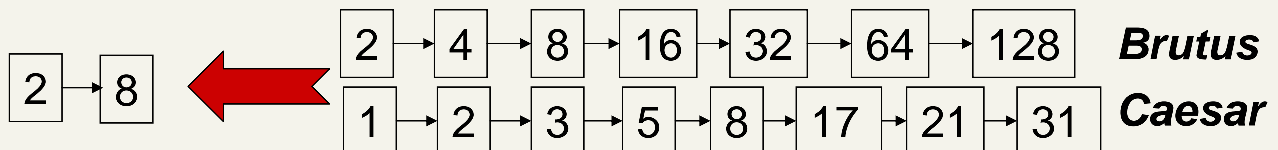
Other stemmers

- Other stemmers exist, e.g., Lovins stemmer http://www.comp.lancs.ac.uk/computing/research/stemming/general/lovins.htm
- Single-pass, longest suffix removal (about 250 rules)
- Motivated by Linguistics as well as IR
- Full morphological analysis - modest benefits for retrieval

Faster postings merges: Skip pointers

Recall basic merge

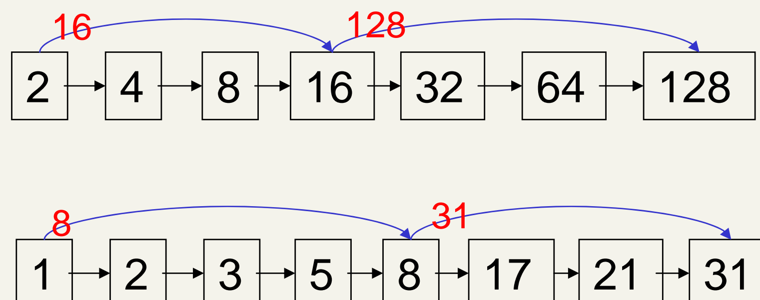
- Walk through the two postings simultaneously, in time linear in the total number of postings entries



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

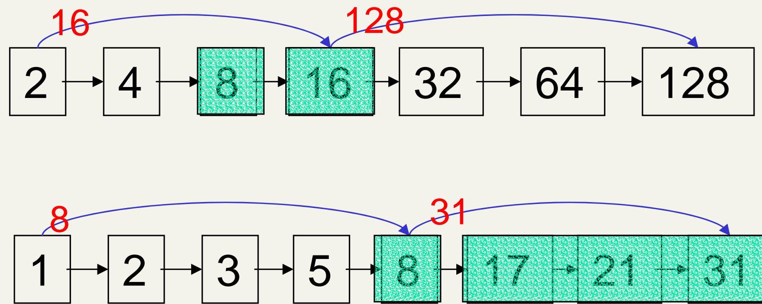
Can we do better?
Yes, if index isn't changing too fast.

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



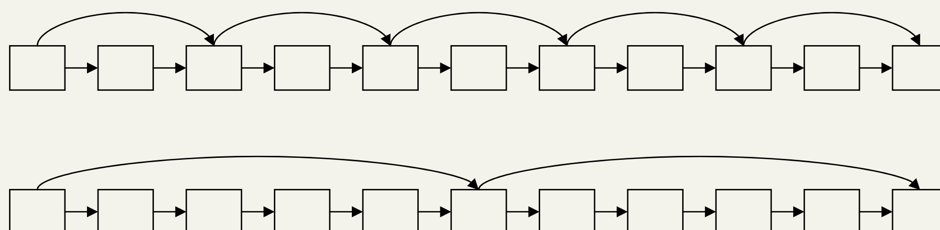
Suppose we've stepped through the lists until we process **8** on each list.

When we get to **16** on the top list, we see that its successor is **32**.

But the skip successor of **8** on the lower list is **31**, so we can skip ahead past the intervening postings.

Where do we place skips?

- Tradeoff:
  - More skips → shorter skip spans ⇒ more likely to skip. But lots of comparisons to skip pointers.
  - Fewer skips → few pointer comparison, but then long skip spans ⇒ few successful skips.



Placing skips

- Simple heuristic: for postings of length L, use $\sqrt{L}$ 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.

Phrase queries

Phrase queries

- Want to answer queries such as ***stanford university*** – as a phrase
- Thus the sentence “I went to university at Stanford” is not a match.
- 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 and Countrymen” would generate the biwords
  - ***friends romans***
  - ***romans and***
  - ***and countrymen***
- Each of these 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:

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.

Extended biwords

- Parse the indexed text and perform part-of-speech-tagging (POST).
- Bucket the terms into (say) Nouns (N) and articles/prepositions (X).
- Now deem any string of terms of the form NX*N to be 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

- Given a query, parse it into N's and X's
  - Segment query into enhanced biwords
  - Look up index
- Issues
  - Parsing longer queries into conjunctions
  - E.g., the query ***tangerine trees and marmalade skies*** is parsed into
  - ***tangerine trees*** *AND* ***trees and marmalade*** *AND* ***marmalade skies***

Other issues

- False positives, as noted before
- Index blowup due to bigger dictionary

Positional indexes

- Store, for each ***term***, entries of the form:

  <number of docs containing ***term***;

  doc1: position1, position2 … ;

  doc2: position1, position2 … ;

  etc.>

Positional index example

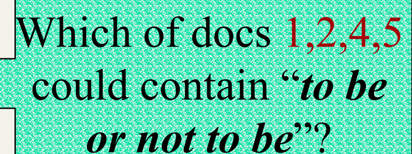
<***be***: 993427;

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

2: 3, 149;

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

5: 363, 367, …>



- Can compress position values/offsets
- Nevertheless, this expands postings storage *substantially*

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

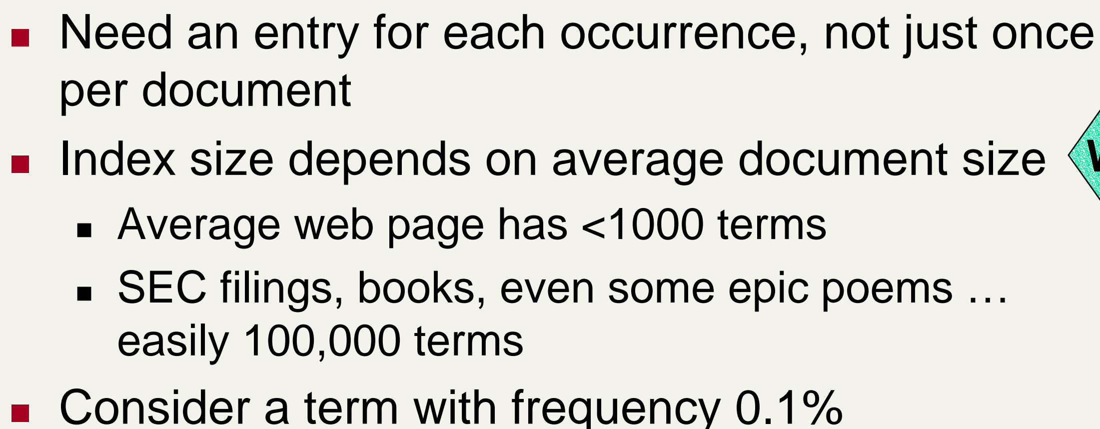
Proximity queries

- LIMIT! /3 STATUTE /3 FEDERAL /2 TORT
  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*?

Positional index size

- Can compress position values/offsets as we did with docs in the last lecture
- Nevertheless, this expands postings storage *substantially*

Positional index size



- Need an entry for each occurrence, not just once per document
- Index size depends on average document size **Why?**
  - 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
1000	1	1
100,000	1	100

Rules of thumb

- Positional index size factor of 2-4 over non-positional index
- Positional index size 35-50% of volume of original text
- Caveat: all of this holds for “English-like” languages

Wild-card queries

Wild-card queries: *

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

  Now retrieve all words in range: ***nom*** ≤ ***w < non****.*

Exercise: from this, how can we enumerate all terms meeting the wild-card query ***pro*cent*** ?

Query processing

- At this point, we have an enumeration of all terms in the dictionary that match the wild-card query.
- We still have to look up the postings for each enumerated term.
- E.g., consider the query:

  se*ate *AND* ***fil*er***

  This may result in the execution of many Boolean *AND* queries.

Permuterm index

- For term ***hello*** index under:
  - ***hello$, ello$h, llohe, lohel, o$hell***

  where $ is a special symbol.
- Queries:
  - **X** lookup on **X$** **X*** lookup on **X*$**
  - ***X** lookup on **X$*** ***X*** lookup on **X***
  - **X*Y** lookup on **Y$X*** **X*Y*Z** ???

  Exercise!

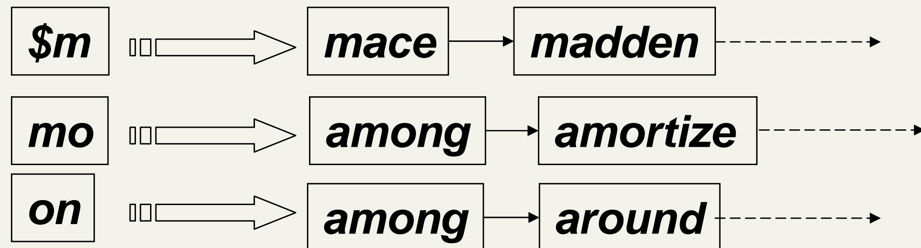
Bigram indexes

- *Permuterm problem: ≈ quadruples lexicon size*
- Another way: index all *k*-grams occurring in any word (any sequence of *k* chars)
- *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
- Index retrieves matching *dictionary terms.*

Bigram indexes - example



Processing *n*-gram wild-cards

- Query ***mon**** can now be run as
  - ***$m** AND **mo** AND **on***
- Fast, space efficient, returns terms that are then run against the dictionary.
- But we'd enumerate ***moon***.
- Must post-filter these terms against query.

Processing wild-card queries

- As before, we must execute a Boolean query for each enumerated, filtered term.
- Wild-cards can result in expensive query execution
  - Avoid encouraging "laziness" in the UI:

Resources for this lecture

- *Managing Gigabytes*, Chapter 3.2, 3.6, 4.3
- *Modern Information Retrieval,* Chapter 8.2
- Shakespeare: http://www.rhymezone.com/shakespeare/
- Try the neat browse by keyword sequence feature!
- Porter's stemmer: http//www.sims.berkeley.edu/~hearst/irbook/porter.html
- H.E. Williams, J. Zobel, and D. Bahle, "Fast Phrase Querying with Combined Indexes", ACM Transactions on Information Systems.

  http://www.seg.rmit.edu.au/research/research.php?author=4