Introduction to Information Retrieval and Boolean model

Reference: Introduction to Information Retrieval
by C. Manning, P. Raghavan, H. Schutze

Structured vs unstructured data

- Structured data tends to refer to information in "tables"

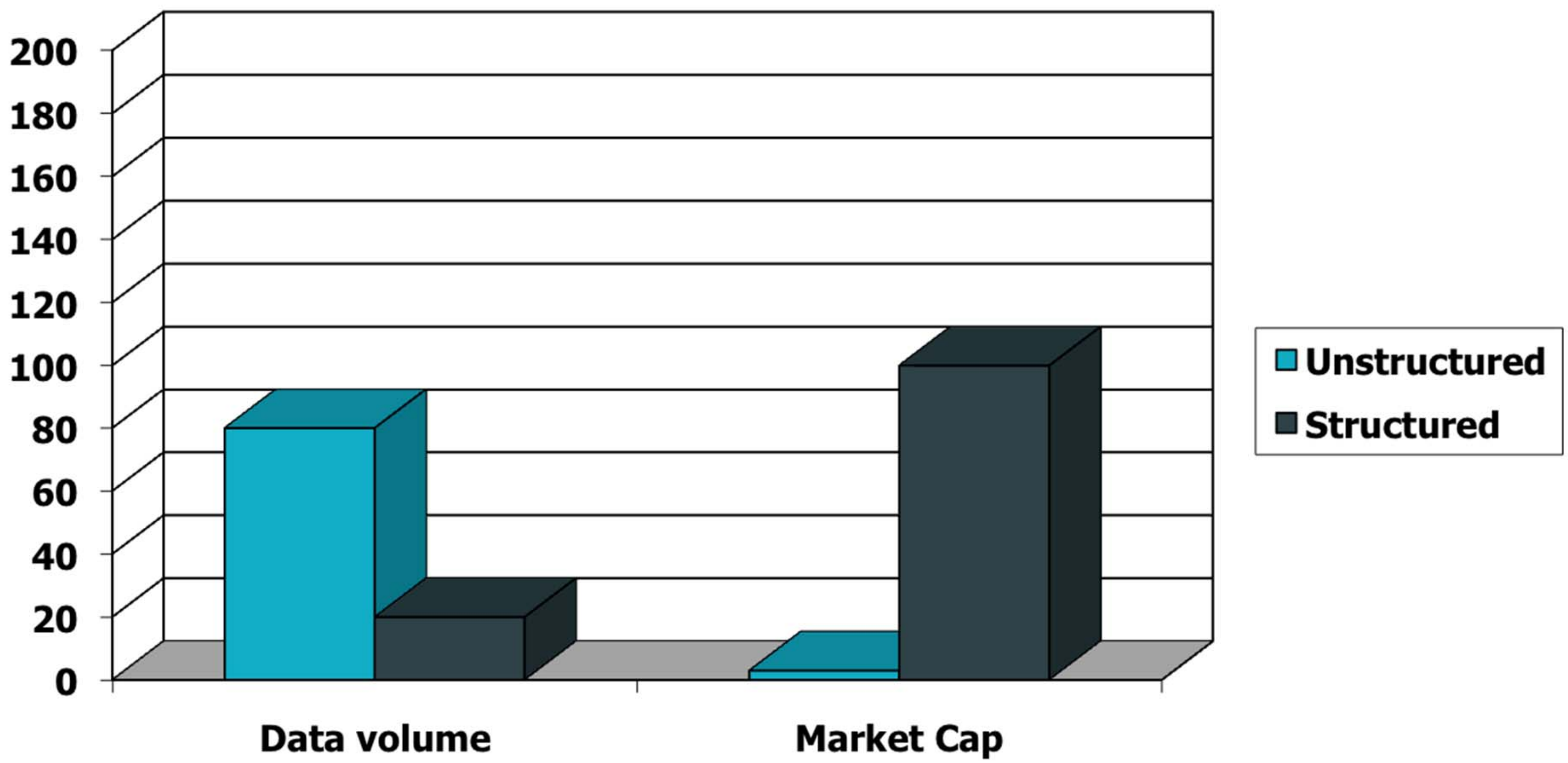
Employee	Manager	Salary
Smith	Jones	50000
Chang	Smith	60000
Ivy	Smith	50000

Typically allows numerical range and exact match (for text) queries, e.g.,
Salary < 60000 AND Manager = Smith.

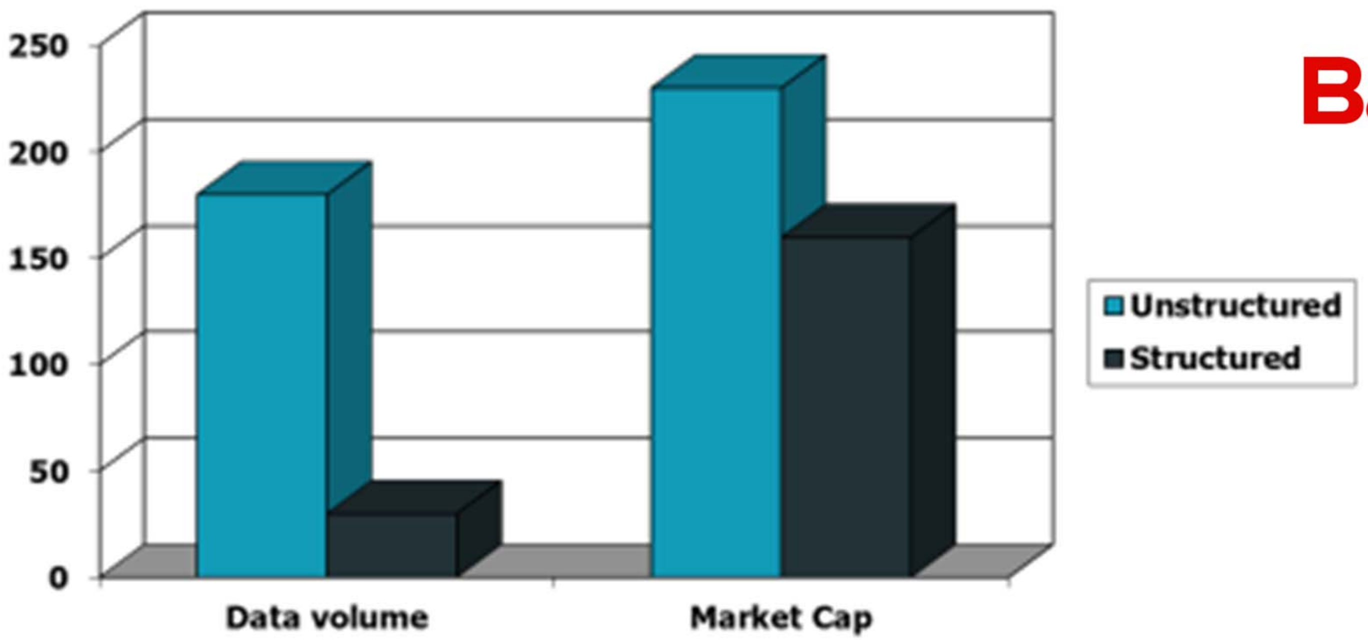
Unstructured data

- Typically refers to free text
- Allows
  - Keyword queries including operators
  - More sophisticated “concept” queries e.g.,
    - find all web pages dealing with *drug abuse*
- Classic model for searching text documents

Unstructured (text) vs. structured (database) data in late nineties

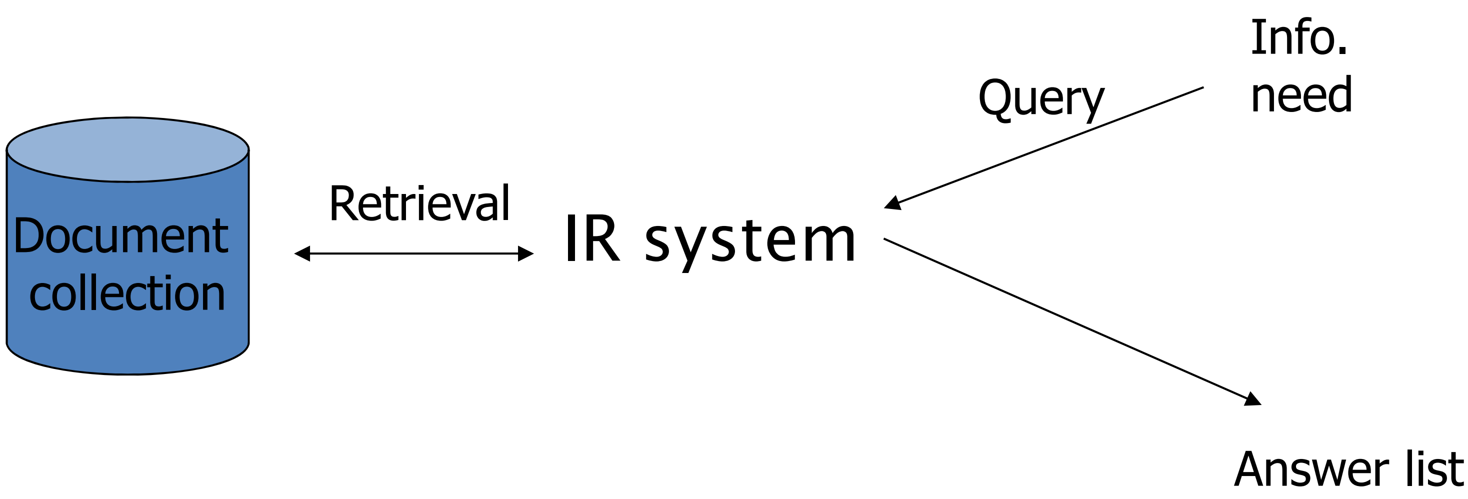


Unstructured (text) vs. structured (database) data now

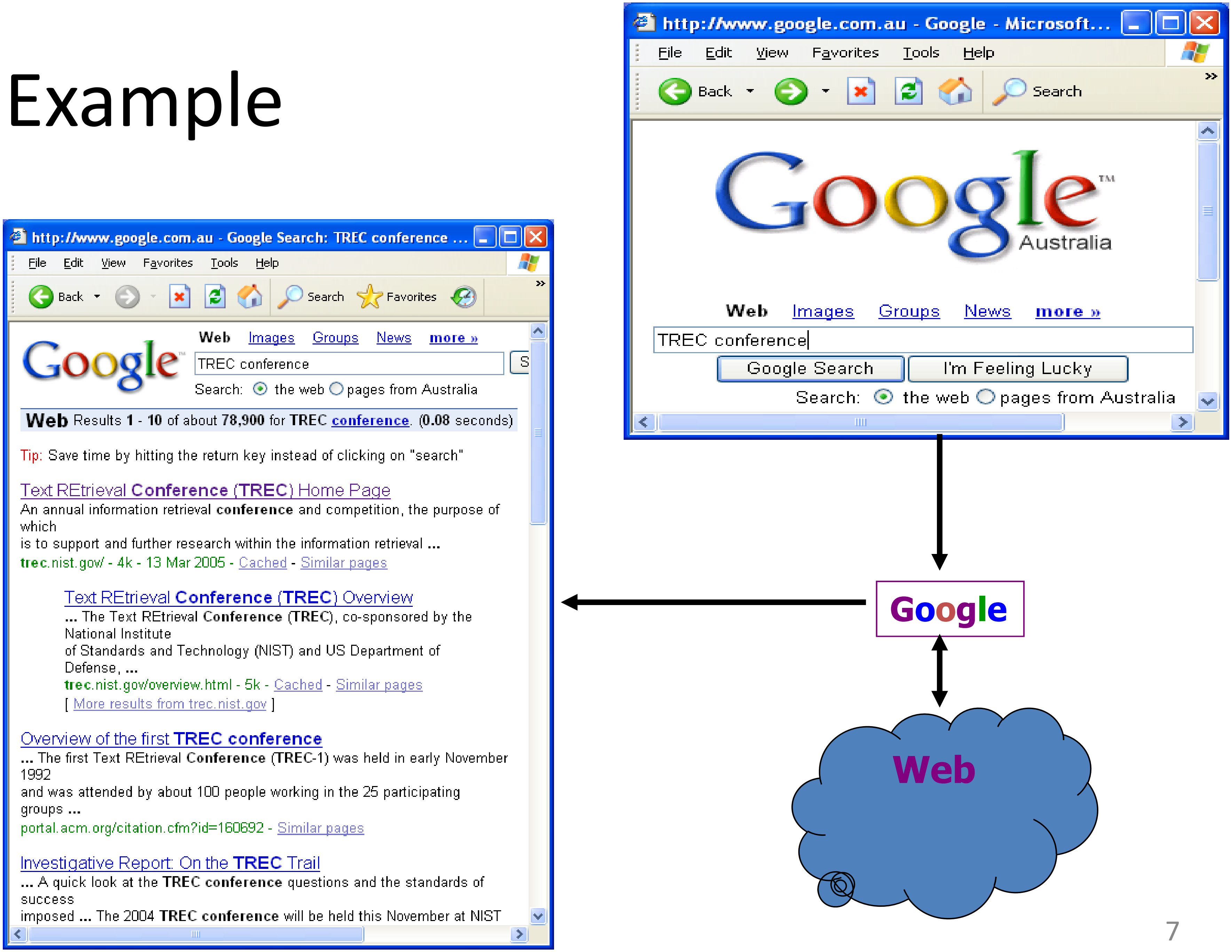


Goal of IR

- Collection: A set of documents
- Goal: Find documents relevant to user's information need



Example


http://www.google.com.au - Google - Microsoft...
File Edit View Favorites Tools Help
Back Search
Google Australia
Web Images Groups News more »
TREC conference
Google Search I'm Feeling Lucky
Search: the web pages from Australia
Google
Web
http://www.google.com.au - Google Search: TREC conference ...
File Edit View Favorites Tools Help
Back Search Favorites
Web Images Groups News more »
TREC conference
Search: the web pages from Australia
Web Results 1 - 10 of about 78,900 for TREC conference. (0.08 seconds)
Tip: Save time by hitting the return key instead of clicking on "search"
Text REtrieval Conference (TREC) Home Page
An annual information retrieval conference and competition, the purpose of which
is to support and further research within the information retrieval ...
trec.nist.gov/ - 4k - 13 Mar 2005 - Cached - Similar pages
Text REtrieval Conference (TREC) Overview
... The Text REtrieval Conference (TREC), co-sponsored by the National Institute
of Standards and Technology (NIST) and US Department of Defense, ...
trec.nist.gov/overview.html - 5k - Cached - Similar pages
[More results from trec.nist.gov]
Overview of the first TREC conference
... The first Text REtrieval Conference (TREC-1) was held in early November 1992
and was attended by about 100 people working in the 25 participating groups ...
portal.acm.org/citation.cfm?id=160692 - Similar pages
Investigative Report: On the TREC Trail
... A quick look at the TREC conference questions and the standards of success
imposed ... The 2004 TREC conference will be held this November at NIST

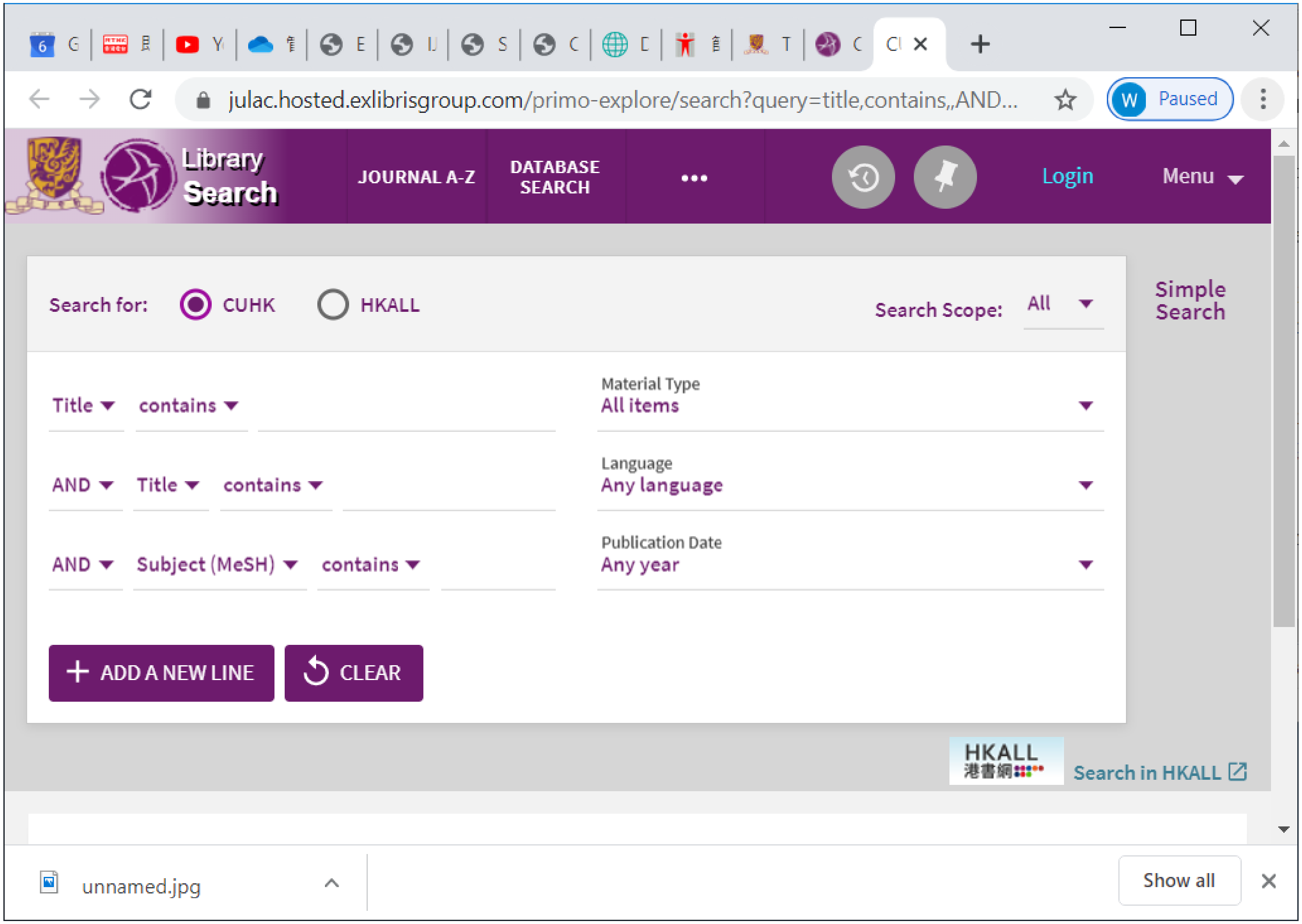
Boolean Model for IR

- Queries are Boolean expressions.
  - e.g., Caesar AND Brutus
- The search engine returns all documents that satisfy the Boolean expression.

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 many years
- Professional searchers still like Boolean queries:
  - You know exactly what you're getting.

Example: Library Search



Boolean Model

- Long, precise queries; proximity operators; incrementally developed; not like web search

A Simple Example

- Consider a document collection of Shakespeare plays
- Which plays of Shakespeare contain the words ***Brutus*** *AND* ***Caesar*** but *NOT* ***Calpurnia***?

Retrieval for Shakespeare Document Collection

- 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

Query: Brutus AND Caesar but NOT Calpurnia

	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 document 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 document collections

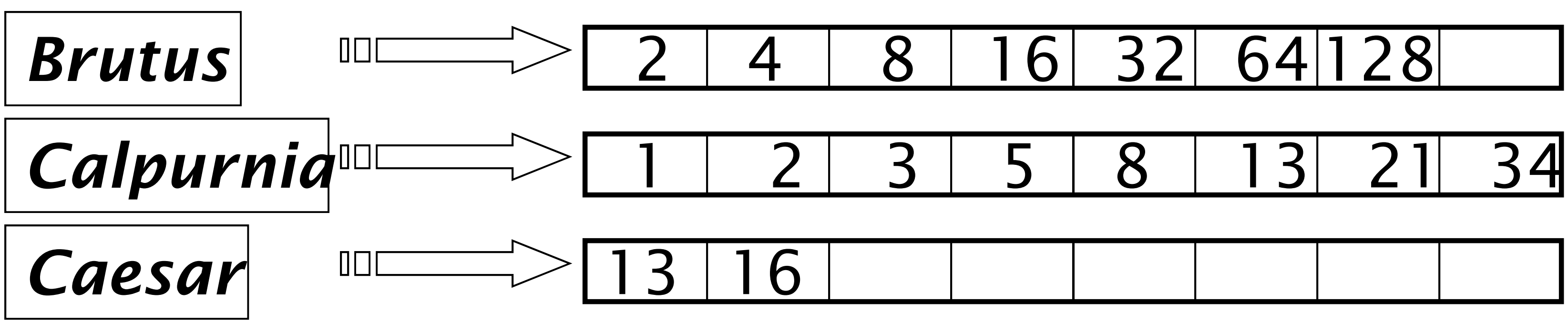
- Consider *N* = 1 million documents, each with about 1K terms.
- Avg 6 bytes/term including spaces/punctuation
  – 6GB of data in the documents.
- Say there are *M* = 500K *distinct* terms among these.

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*: store a list of all documents that contain *T*.
- Each document is identified by a document ID



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

Inverted Index

- A fundamental structure that can support various kinds of IR models including Google search model.

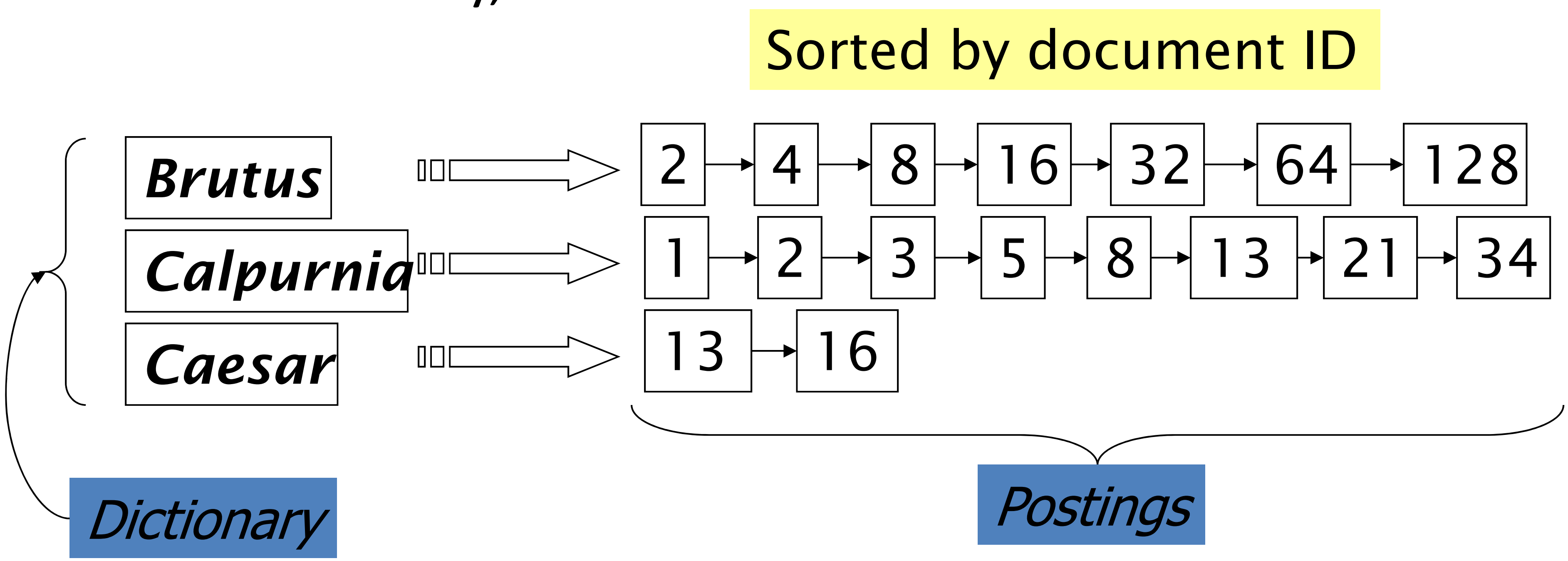
https://www.google.com/search/howsearchworks
movie: Trillions of Questions, No Easy Answers

6:10 – 8:00 – senior staff

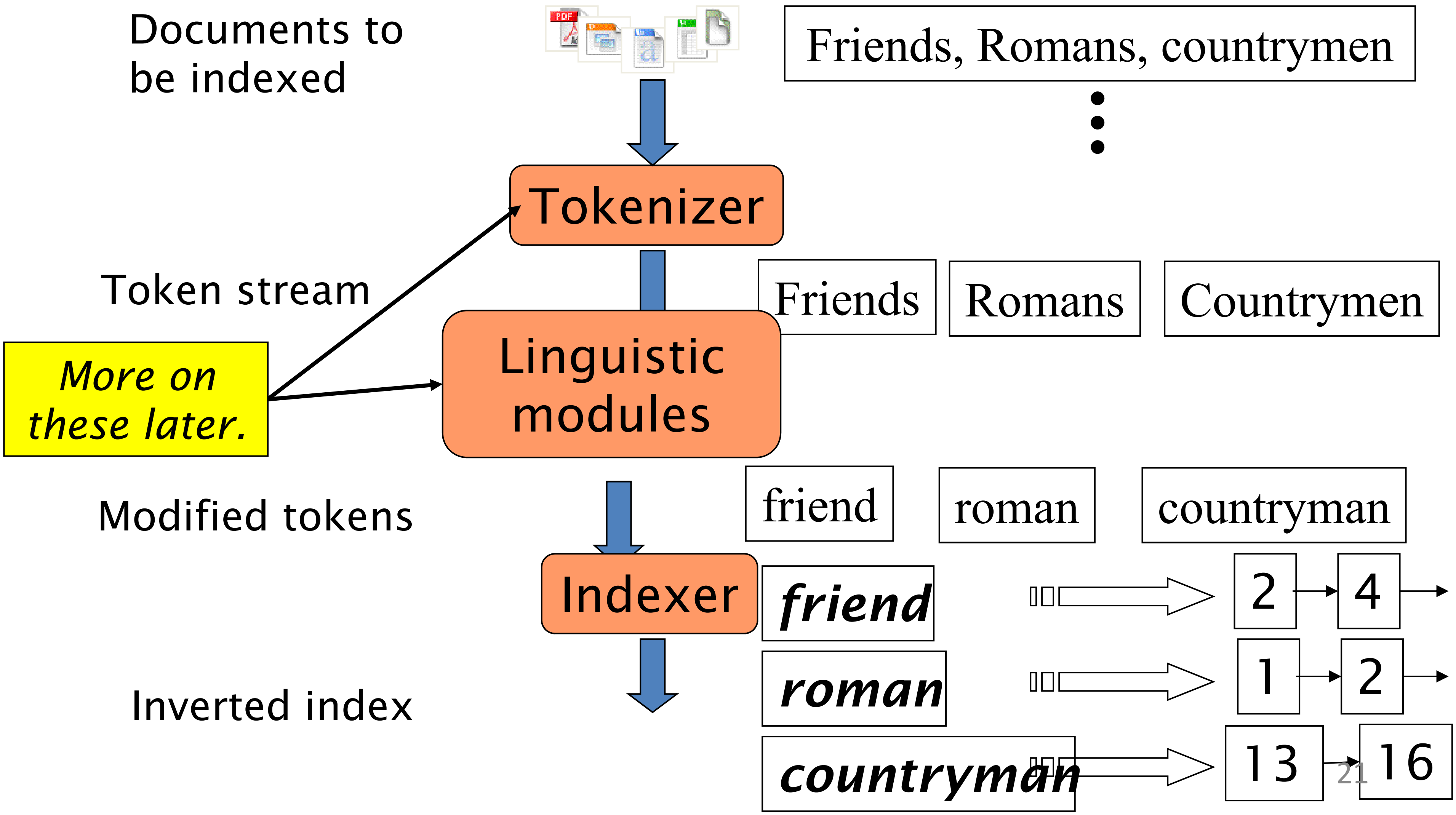
22:55-25:00 - indexing

Inverted index

- Use a variable-sized posting lists
  - Dynamic space allocation
  - Insertion of terms into documents easy
  - In memory, can use linked lists



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

Indexer steps

- 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

Indexer steps

- 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 frequency

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

total term frequency

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

term frequency

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

Query processing

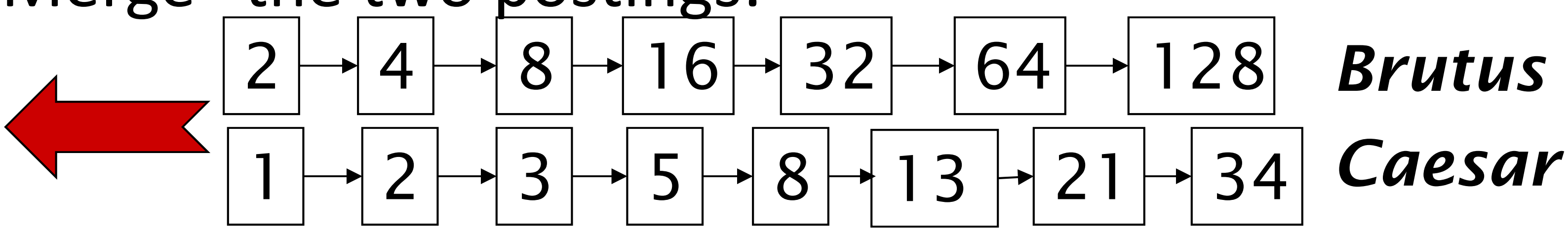
- Consider processing the query:

  Brutus *AND* ***Caesar***

  – Locate ***Brutus*** in the Dictionary;
    - Retrieve its postings.

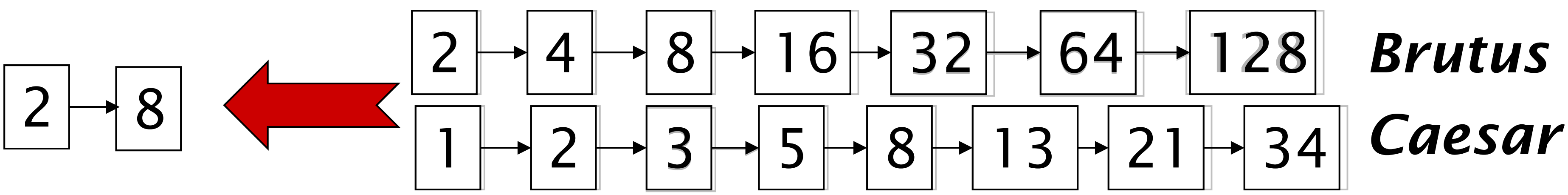
  – Locate *Caesar* in the Dictionary;
    - Retrieve its postings.

  – "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.

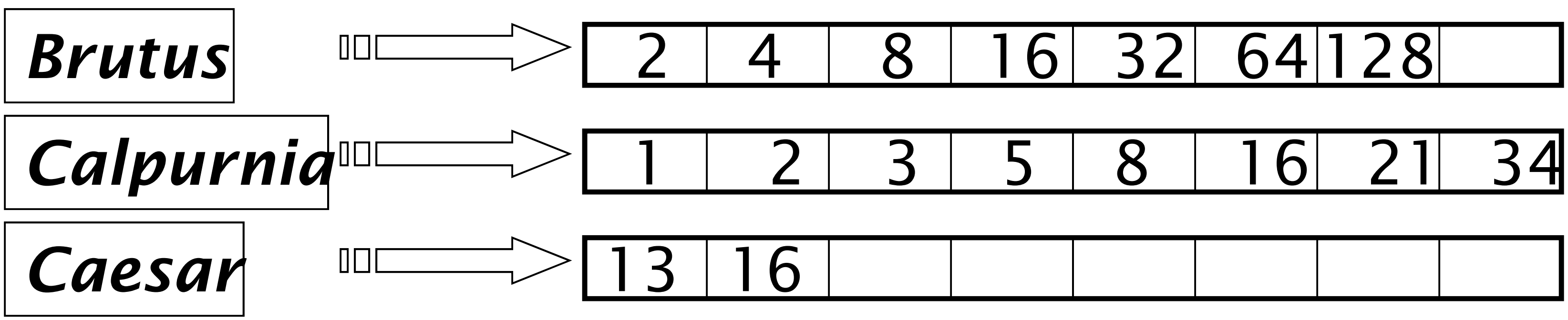
Basic postings intersection

- A "merge" algorithm

```
INTERSECT(p1, p2)
answer ← ⟨ ⟩
while p1 ≠ NIL and p2 ≠ NIL
do if docID(p1) = docID(p2)
     then ADD(answer, docID(p1))
          p1 ← next(p1)
          p2 ← next(p2)
     else if docID(p1) < docID(p2)
            then p1 ← next(p1)
            else p2 ← next(p2)
return answer
```

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.

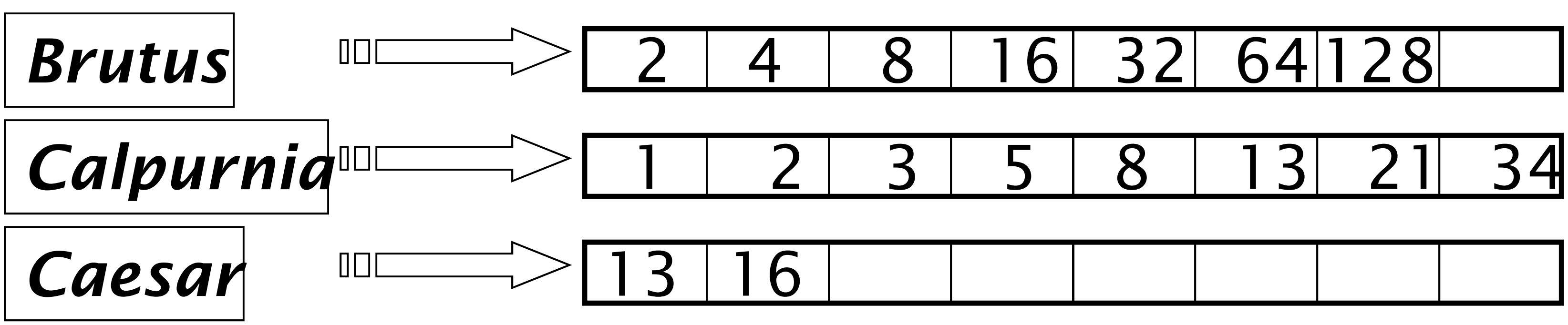


Query: ***Brutus*** *AND* ***Calpurnia*** *AND* ***Caesar***

Query optimization example

- Process in order of increasing *document frequency (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***.

Query optimization

```
INTERSECT(⟨t_1, ..., t_n⟩)
terms ← SORTBYINCREASINGFREQUENCY(⟨t_1, ..., t_n⟩)
result ← POSTINGS(FIRST(terms))
terms ← REST(terms)
while terms ≠ NIL and result ≠ NIL
do list ← POSTINGS(FIRST(terms))
    result ← INTERSECT(result, POSTINGS(FIRST(terms)))
    terms ← REST(terms)

return result
```

► **Figure 1.8** Algorithm for conjunctive queries that returns the set of documents containing each term in the input list of terms.

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.

Phrase queries

- We want to be able to answer queries such as ***"air conditioner"*** – as a phrase
- Thus the sentence *"After washing my hair with this conditioner, I dry my hair with hot air"* 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
- ***air conditioner filter system*** can be broken into the Boolean query on biwords:

air conditioner *AND* ***conditioner filter*** *AND* ***filter system***

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:

<***term: termID;***

doc1: position1, position2 … ;

doc2: position1, position2 … ;

:

>

Example:

<***to***: 993427;

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

2: 3, 149;

4: 8, 16, 190, 429, 433;

5: 363, 367, …>

Positional index example

- 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***: 993427;
1: 7, 18, 33, 72, 86, 231;
2: 3, 149;
4: 8, 16, 190, 429, 433;
5: 363, 367;…>
<***be***: 178239;
1: 17, 25;
4: 17, 191, 291, 430, 434;
5: 14, 19, 101; …>

Which of docs 1,2,4,5 could contain "***to be or not to be***"?

Answer: 4

Proximity queries

- Same general method for proximity searches
- Within k word proximity search

  e.g. employment /3 place

  /*k* means "within *k* words of".
- The algorithm for "merge" two posting lists can be extended to handle within k word proximity search
- Clearly, positional indexes can be used for such queries; biword indexes cannot.

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.

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 ¼ of the time of using just a positional index
  - It required 26% more space than having a positional index alone

Semi-structured data

- But in fact almost no data is “unstructured”
- E.g., this slide has distinctly identified zones such as the *Title* and *Bullets*
- Facilitates “semi-structured” search such as
  - *Title* contains data AND *Bullets* contain search