Introduction to

Information Retrieval

Introducing Information Retrieval and Web Search

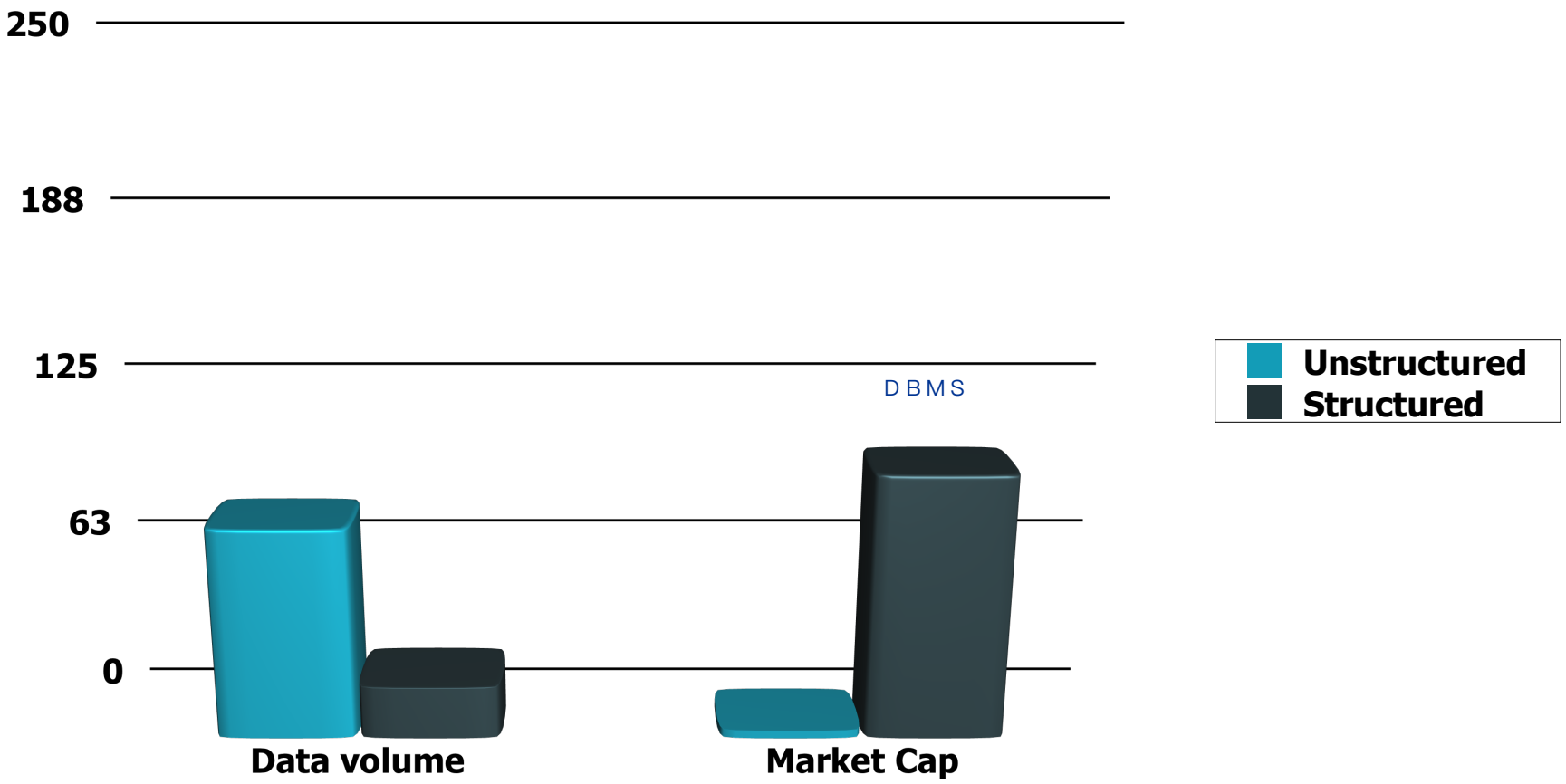
Information Retrieval

從大量資料裡面，萃取出和輸入的query最相關的結果

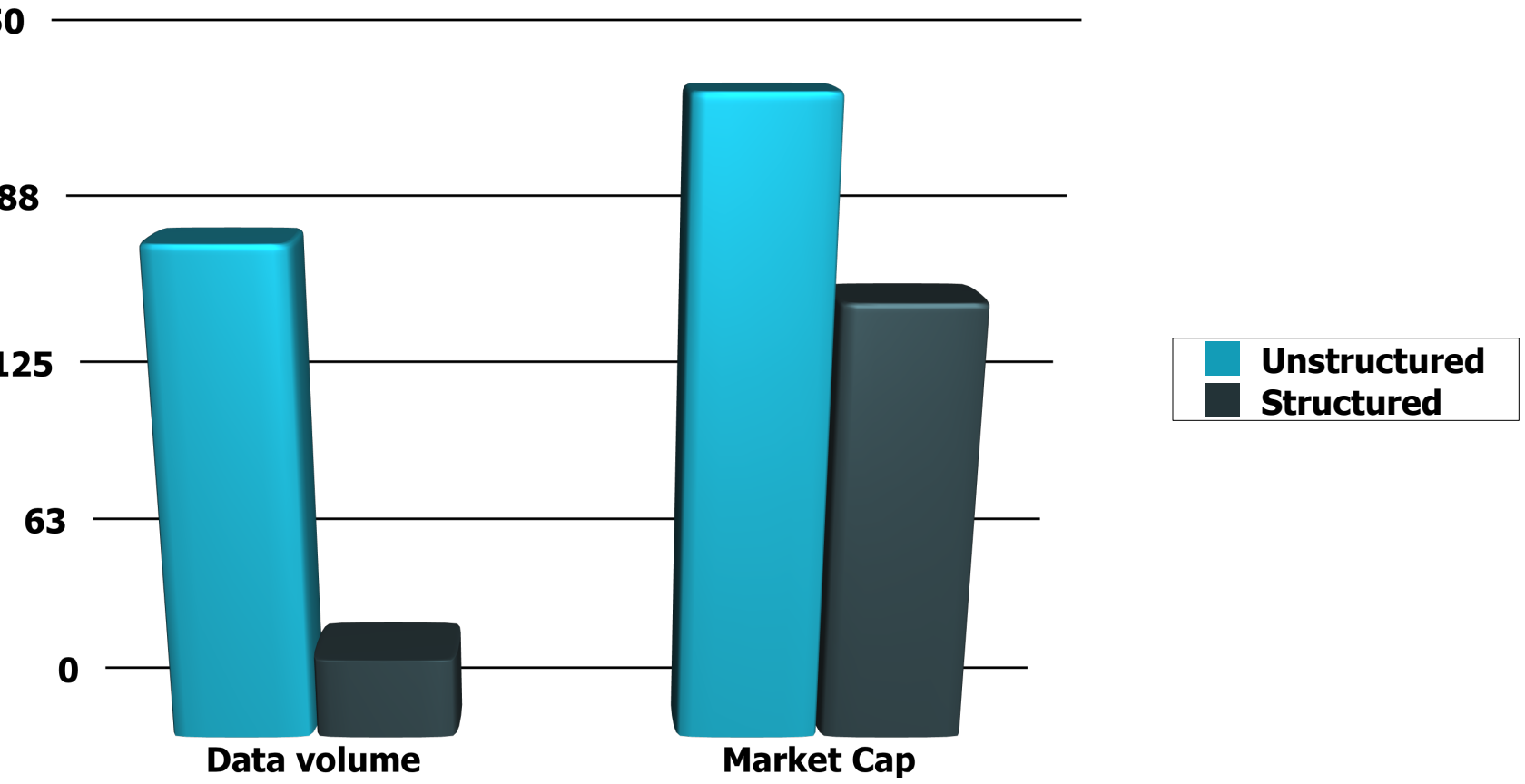
- Information Retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers).
  - These days we frequently think first of web search, but there are many other cases:
    - E-mail search
    - Searching your laptop
    - Corporate knowledge bases
    - Legal information retrieval

Unstructured (text) vs. structured (database) data in the mid-nineties

結構化的資料在商業上比較有用



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

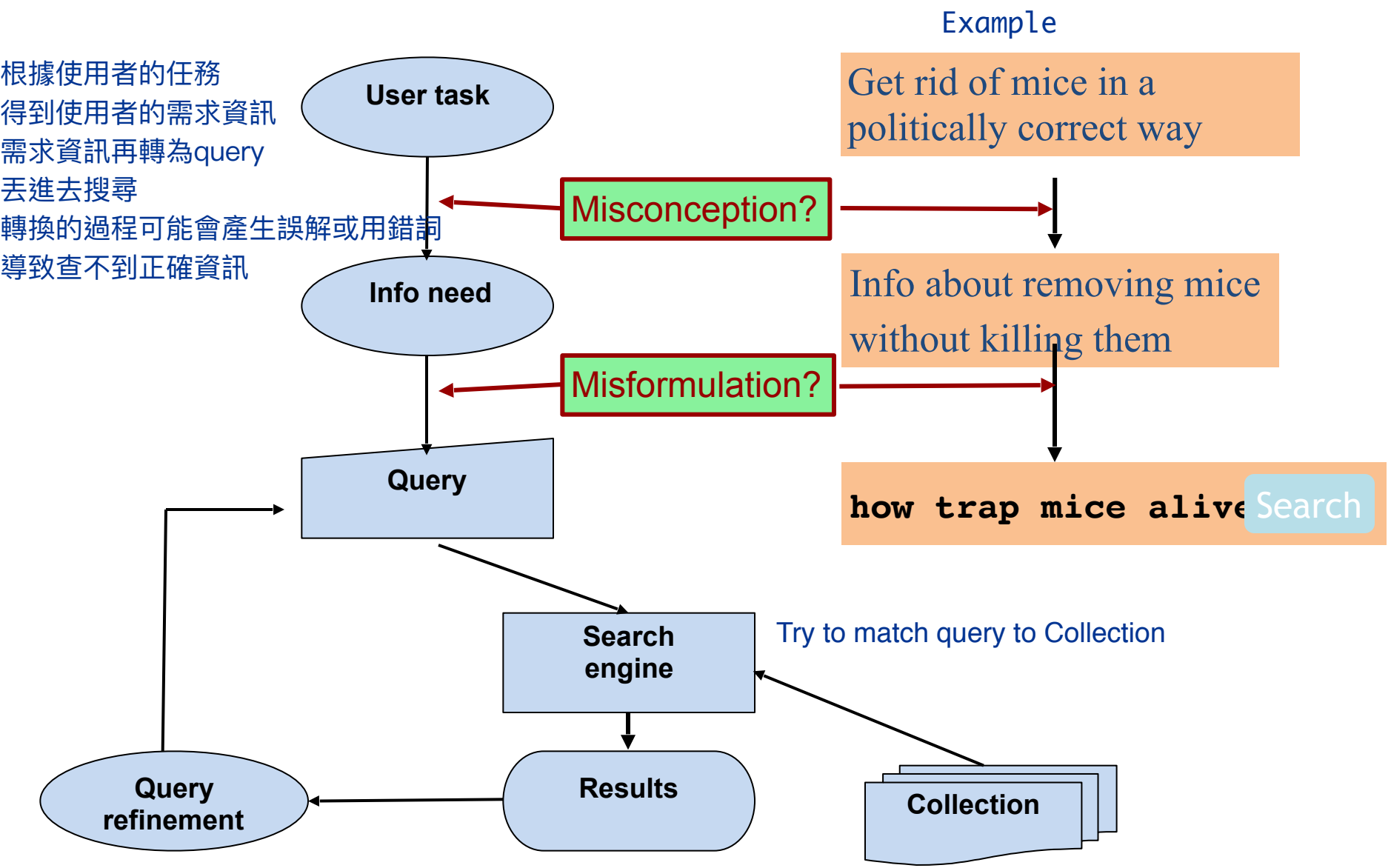


Basic assumptions of Information Retrieval

- Collection: A set of documents
  - Assume it is a static collection for the moment

- Goal: Retrieve documents with information that is relevant to the user's information need and helps the user complete a task

The classic search model



How good are the retrieved docs?

查到的文件有多少是相關的？

- *Precision* : Fraction of retrieved docs that are relevant to the user's information need

有多少相關的文件被查到了？

- *Recall* : Fraction of relevant docs in collection that are retrieved

  - More precise definitions and measurements to follow later

Introduction to

Information Retrieval

Term-document incidence matrices

Unstructured data in 1620

- 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 that not the answer?
  - Slow (for large corpora)
  - ***NOT Calpurnia*** is non-trivial
  - Other operations (e.g., find the word ***Romans*** near ***countrymen***) not feasible
  - Ranked retrieval (best documents to return)
    - Later lectures

要搜尋包含Brutus, Caesar但不包含Calpurnia的文件
如果先找出包含Brutus和Caesar的所有，再把包含Calpurnia的刪掉
這樣很花時間，也很難做，無法Ranking

Term-document incidence matrices

類似布林事件表格
只記錄某term有沒有出現在某document中

Document/term	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

Brutus *AND* ***Caesar*** *BUT NOT* ***Calpurnia***

Incidence vectors

有了剛剛的表格後，就可以用bit operation算出結果

- 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**

Complemented = NOT operator

	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

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 collections

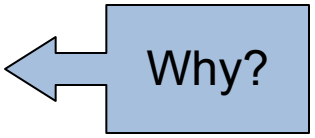
Too large and sparse

- Consider *N* = 1 million documents, each with about 1000 words.
- Avg 6 bytes/word 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.

Document太多，導致word太多
會變成一個非常稀疏的矩陣，很沒有效率，也浪費空間
解法：用inverted index



- 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.

Introduction to

Information Retrieval

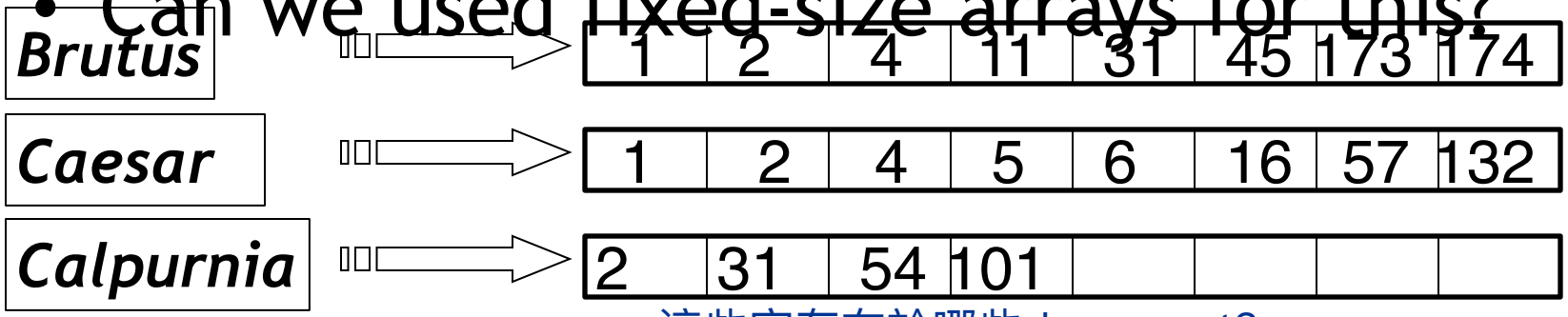
The Inverted Index

The key data structure underlying modern IR

Inverted index

Only key concepts instead of common words

- For each term *t*, we must store a list of all documents that contain *t*.
  - Identify each doc by a **docID**, a document serial number
- Can we used fixed-size arrays for this?



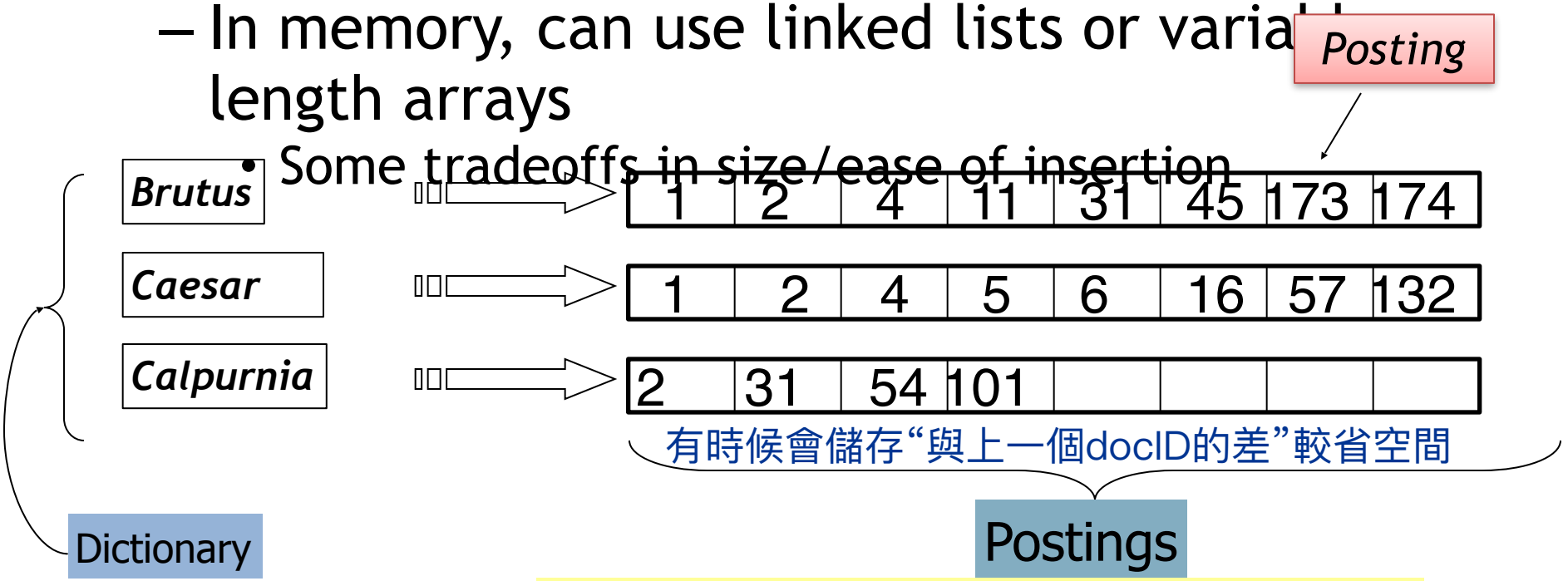
這些字存在於哪些document?

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

Inverted index

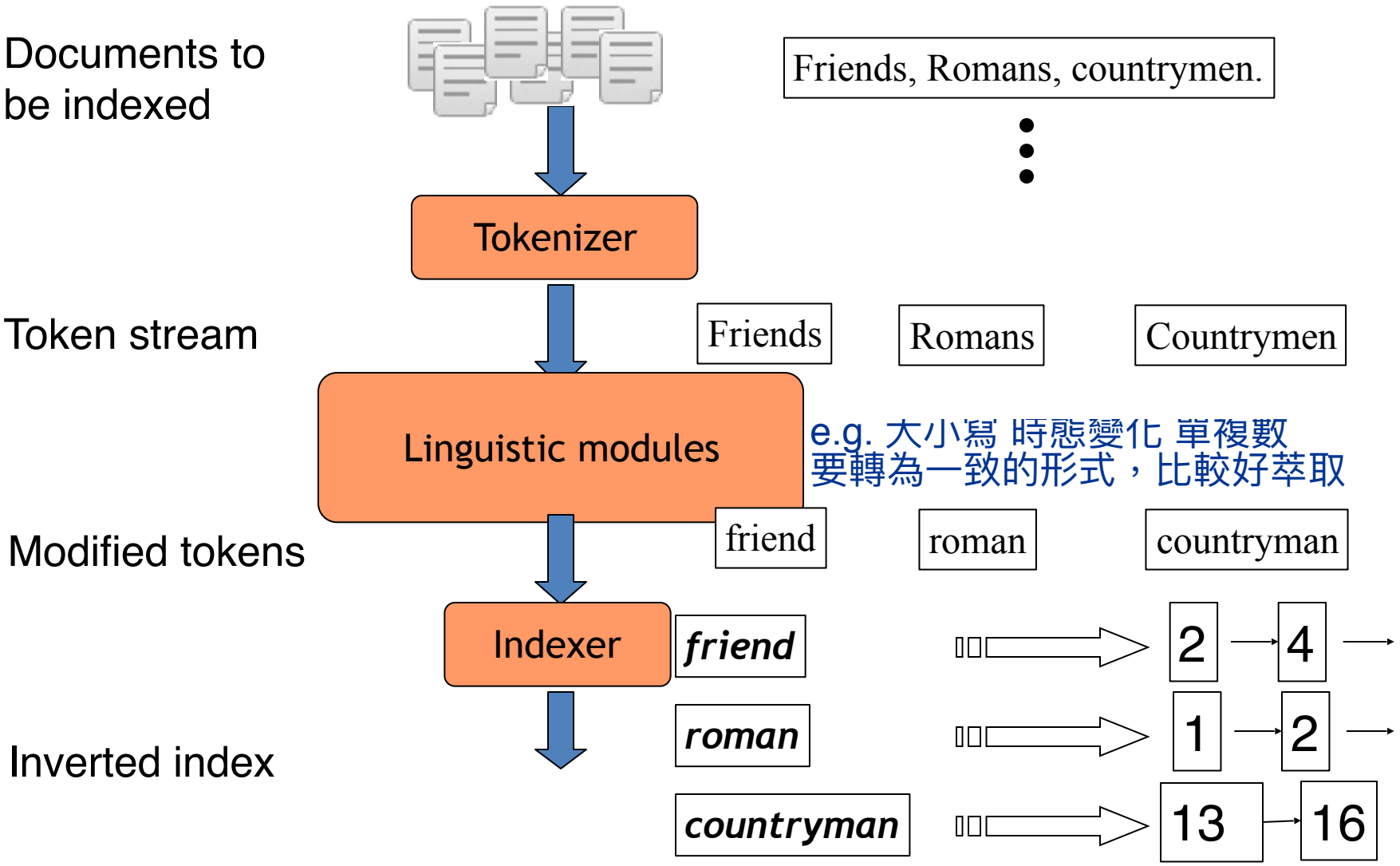
- We need variable-size postings lists
  - On disk, a continuous run of postings is normal and best
  - In memory, can use linked lists or variable length arrays
    - Some tradeoffs in size/ease of insertion

像LINKED LIST



Sorted by docID (more later on why).

Inverted index construction



Initial stages of text processing

- Tokenization 把文本切成一個個單字，比較容易搜尋
  - Cut character sequence into word tokens
    - Deal with ***"John's"***, ***a state-of-the-art solution***
- Normalization 需要改成一樣的形式
  - Map text and query term to same form
    - You want ***U.S.A.*** and ***USA*** to match
- Stemming 動詞變化、時態變化、單複數…
  - We may wish different forms of a root to match
    - ***authorize***, ***authorization***
- Stop words 太常見的連接詞、介系詞等可以忽略
  - We may omit very common words (or not)
    - ***the***, ***a***, ***to***, ***of***

Indexer steps: Token sequence

- Sequence of (Modified token, Document ID) pairs.

儲存每一個字
出現在哪個docID裡
(key, value) = (term, docID)
第一階段key可以重複
就像Hadoop的Mapper一樣

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

Sorted by docID

Term	docID
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

- Sort by terms
  - And then docID

1st sort: 單字排序
2nd sort: docID

Core indexing step

Term	docID
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	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

Indexer steps: Dictionary & Postings

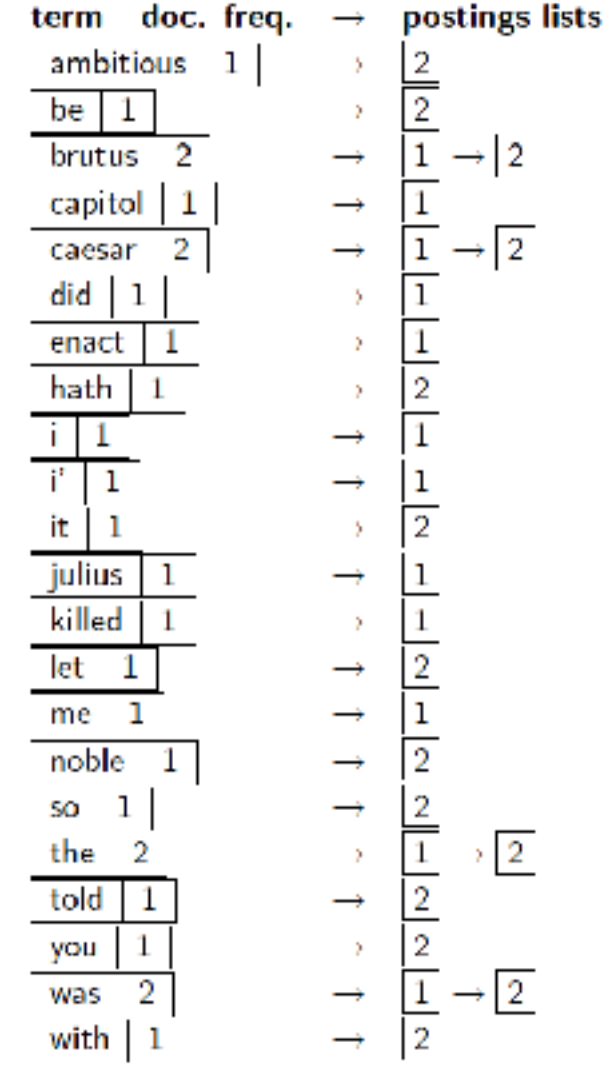
- Multiple term entries in a single document are merged.
- Split into Dictionary and Postings
- Doc. frequency information is added.

因為已經排序好了
所以在這個階段轉為Postings Lists就很方便
有點像Mapper輸出時會自動排序好
在Reducer時就可以輕鬆地處理

Why frequency?
Will discuss later.

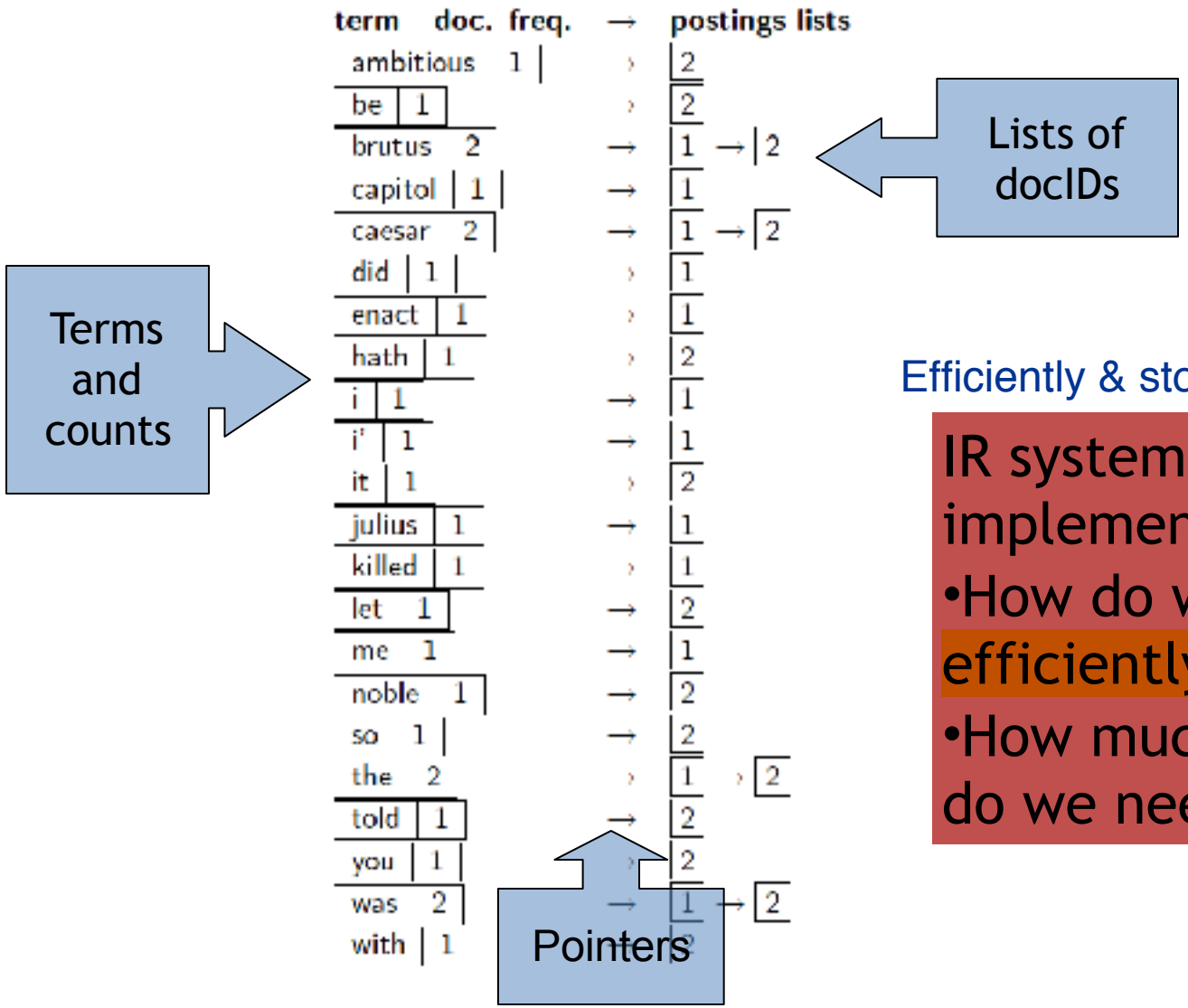
Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
'	1
t	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

Doc freq = 幾個doc有此term



Where do we pay in storage?

Sorted list -> use BS or hash function



Efficiently & storage is tradeoff

IR system implementation

- How do we index efficiently?
- How much storage do we need?

The index we just built

- How do we process a query?
  - Later - what kinds of queries can we process?

Introduction to

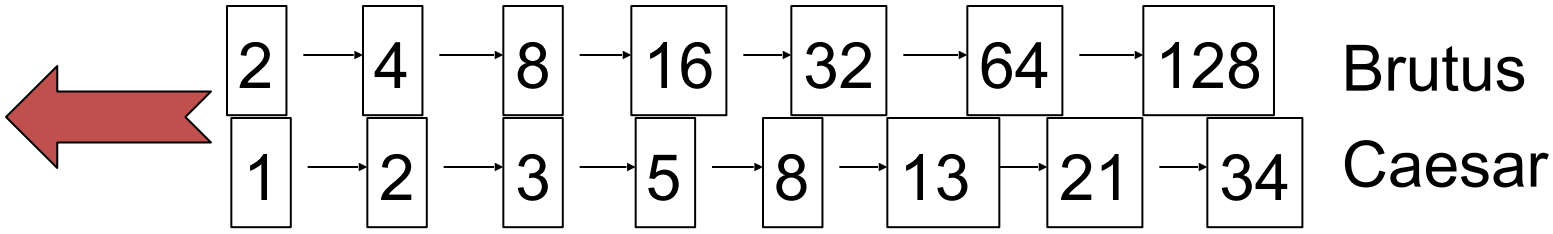
Information Retrieval

Query processing with an inverted index

Query processing: AND

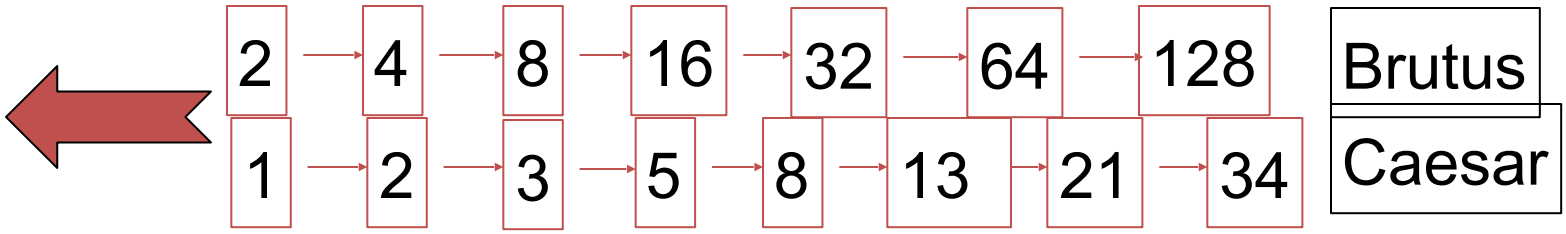
假設要搜尋Brutus AND Caesar
那就回傳這兩個字的Postings Lists並作合併

- 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 (intersect the document sets):



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.

Intersecting two postings lists (a "merge" algorithm)

```
INTERSECT(p1, p2)
 1  answer ← ⟨ ⟩
 2  while p1 ≠ NIL and p2 ≠ NIL
 3  do if docID(p1) = docID(p2)
 4        then ADD(answer, docID(p1))
 5             p1 ← next(p1)
 6             p2 ← next(p2)
 7        else if docID(p1) < docID(p2)
 8                then p1 ← next(p1)
 9                else p2 ← next(p2)
10  return answer
```

Introduction to

Information Retrieval

The Boolean Retrieval Model & Extended Boolean Models

Boolean queries: Exact match

- The Boolean retrieval model is being able to ask a query that is a Boolean expression:
  - Boolean Queries are queries using *AND*, *OR* and *NOT* to join query terms
    - Views each document as a set of words
    - Is precise: document matches condition or not.
  - Perhaps the simplest model to build an IR system on
- Primary commercial retrieval tool for 3 decades.
- Many search systems you still use are Boolean:
  - Email, library catalog, Mac OS X Spotlight

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

- Largest commercial (paying subscribers) legal search service (started 1975; ranking added 1992; new federated search added 2010)
- Tens of 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
    - /3 = within 3 words, /S = in same sentence

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

- Another example query:
  - Requirements for disabled people to be able to access a workplace
  - disabl! /p access! /s work-site work-place (employment /3 place
- Note that SPACE is disjunction, not conjunction!
- Long, precise queries; proximity operators; incrementally developed; not like web search
- Many professional searchers still like Boolean search
  - You know exactly what you are getting
- But that doesn't mean it actually works better….

Boolean queries: More general merges

- Exercise: Adapt the merge for the queries:

  Brutus** AND NOT **Caesar

  Brutus** OR NOT **Caesar

  = –(–B AND – (NOT C))

  = –(–B AND C)

- Can we still run through the merge in time O(*x*+*y*)? What can we achieve?

Merging

What about an arbitrary Boolean formula?

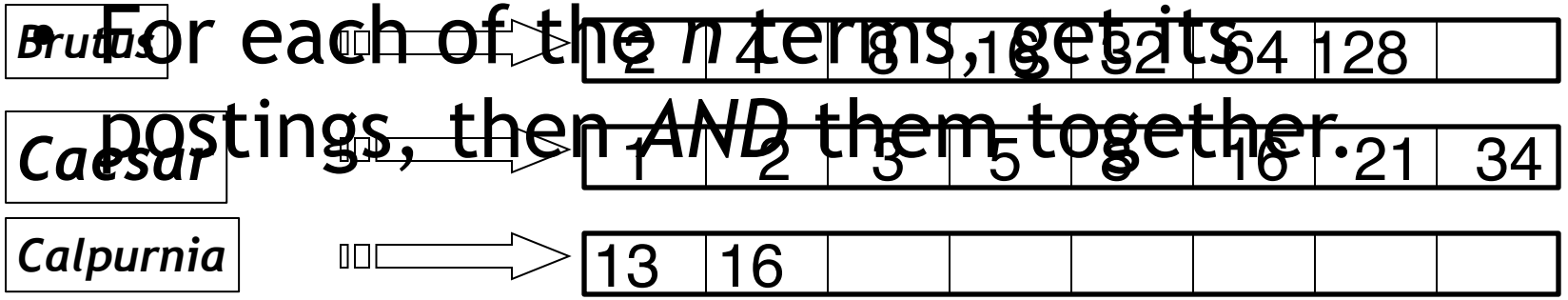
***(Brutus** OR **Caesar)** AND NOT*

(Antony** OR **Cleopatra)

- Can we always merge in "linear" time?
  - Linear in what?
- Can we do better?

Query optimization

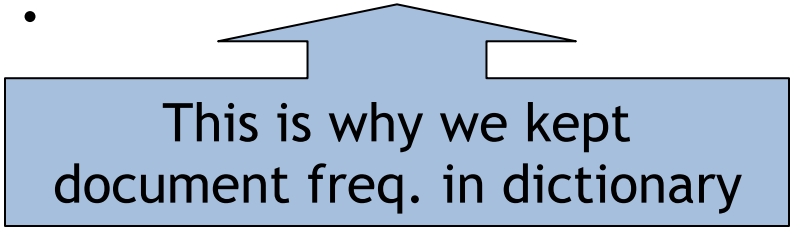
- What is the best order for query processing?
- Consider a query that is an *AND* of *n* terms.
- For each of the *n* terms, get its postings, then *AND* them together.

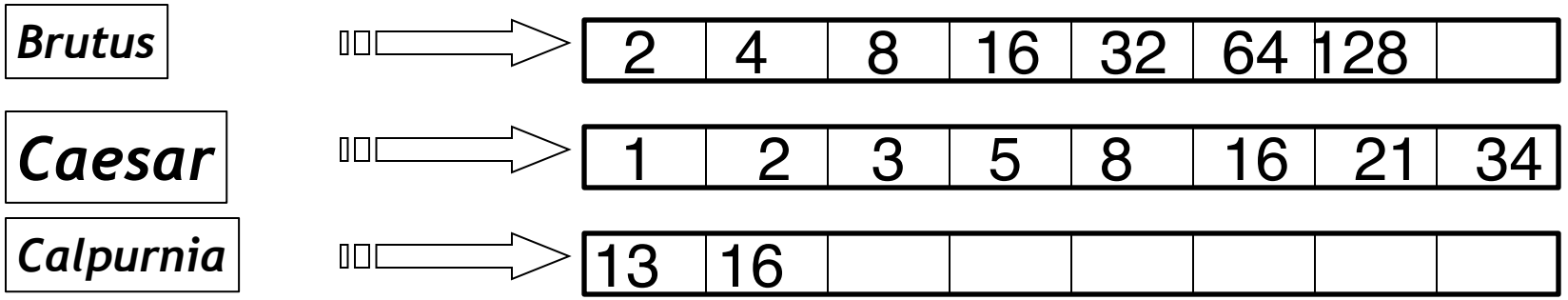


Query: ***Brutus*** *AND* ***Calpurnia*** *AND* ***Caesar***

Query optimization example

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





因為有儲存doc freq, 所以可以從size最短的開始做AND

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

More general optimization

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

- Which two terms should we process first?

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

Query processing exercises

- Exercise: 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: in this, each query term appears only once in the query.

Exercise

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

Introduction to

Information Retrieval

Phrase queries and positional indexes

Phrase queries

兩個單字組成一個字彙的搜尋方式

- 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

因為這樣只能找到同時包含這些term的文章
但不能表示這些term會是一個phrase

雖然說用雙引號可以明確地表達Phrase Quaries
但使用者並不會乖乖地每次都用雙引號

A first attempt: Biword indexes

有點像把原本的資料做Bigram，然後用這些Bigram當作Postings Lists的Term

- 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

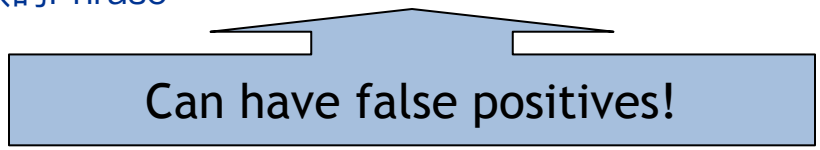
- Longer phrases can be processed by breaking them down
- ***stanford university palo alto*** can be broken into the Boolean query on biwords:

stanford university AND university palo AND palo alto

有幾組biwords，就會有n!個columns(所有可能排列)

Without the docs, we cannot verify that the docs matching the above Boolean query do contain the phrase.

不過即便如此，也不能保證回傳的答案一定是輸入的Phrase
有可能某個文件剛好出現這三組Bi-words
但卻在文章的不同地方



Issues for biword indexes

- False positives, as noted before
- Index blowup due to bigger dictionary
  - Infeasible for more than biwords, big even for them

缺點：Term會太多、不保證完全正確

- Biword indexes are not the standard solution (for all biwords) but can be part of a compound strategy

Solution 2: Positional indexes

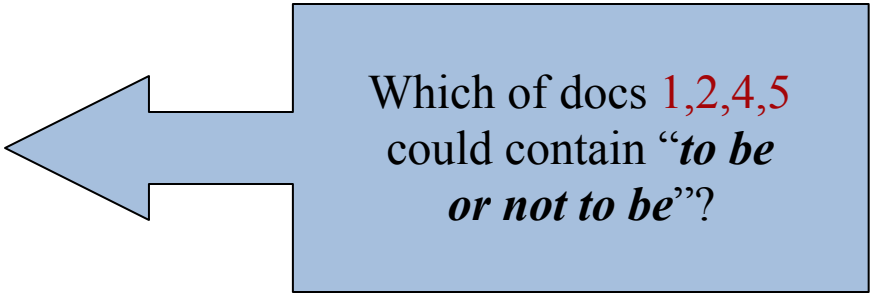
- In the postings, store, for each ***term*** the position(s) in which tokens of it appear:

儲存一個字所在的Doc，以及在每一個存在的Doc的位置

<***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, …>

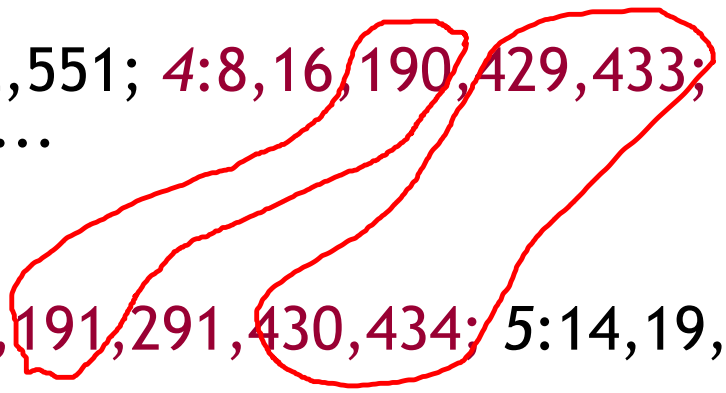


- For phrase queries, we use a merge algorithm recursively at the document level
- But we now need to deal with more than just equality 要多檢查位置

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; ...

文件4中，有三個"to be"



- Same general method for proximity searches

Proximity queries

如果我們可以容忍兩個字之間有一點點距離

Positional Index也可以解決，但用Biwords就不行了

- 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

原本的Postings Lists，Index只要記錄所有文件出現的「唯一字」即可
但Positional Index，「唯一字」的每一次出現都要當成index（？？？？？

- 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%

Why?

Document size	Postings	Positional postings
1000	1	1
100,000	1	100

Combination schemes

如果是經常一起出現的兩個字
例如人名、地名、或文法固定用法等
就可以考慮把這兩個字視為一個字，直接當成Index

- 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

Rules of thumb

- A positional index is 2–4 as large as a non-positional index

  只儲存每個單字的 Doc id –> 只會佔原文的10–15%容量

  若要再儲存每個單字的position –> 佔原文的35%–50%容量
- Positional index size 35–50% of volume of original text

  – Caveat: all of this holds for “English-like” languages

Introduction to

Information Retrieval

Structured vs. Unstructured Data

IR vs. databases: 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

Semi-structured data

- In fact almost no data is “unstructured”
- E.g., this slide has distinctly identified zones such as the *Title* and *Bullets*
  - … to say nothing of linguistic structure
- Facilitates “semi-structured” search such as
  - *Title* contains data AND *Bullets* contain search
- Or even
  - *Title* is about Object Oriented Programming AND *Author* something like stro*rup
  - where * is the wild-card operator