Information Retrieval (IR) and Data Mining (DM)

By: **Dr. LOUNNAS Bilal**

Term-document incidence

Term-document incidence will be the best solution?

Take the vectors for Brutus, Caesar and Calpurnia (complemented)

	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	**1**	**0**	**1**	**1**	**1**	**1**
Cleopatra	1	0	0	0	0	0
mercy	1	0	1	1	1	1
worser	1	0	1	1	1	0
Bitwise AND	1	0	0	1	0	0

1. Antony and Cleopatra
2. Hamlet

Bigger collection

Suppose that :

- Consider N = 1M documents, each with about 1K (1000) terms (words).
- 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

Why?

- 500K x 1M matrix has half-a-trillion (500 000 000 000) 0's and 1's.
- But it has no more than one billion 1's (1 000 000 000).
  - matrix is extremely sparse.

What's a better representation?

We only record the 1's positions. and ignore the 0's positions

Can't build the matrix

Why?

- 500K x 1M matrix has half-a-trillion (500 000 000 000) 0's and 1's.
- But it has no more than one billion 1's (1 000 000 000).
  - matrix is extremely sparse.

What's a better representation?

We only record the 1's positions. and ignore the 0's positions

Inverted index

Inverted index as data structure

Inverted index is the key data structure underlying modern IRs, form systems running on a single laptop to those running in biggest commercial search engine.

- It's DS that exploit the sparsity of Term-document incidence
- Allow a very efficient retrieval.

Inverted index

Inverted index

- For each term **t**, we must store a list of all documents that contain that term **t**.
  - Identify each doc by **docID**, a document serial number.

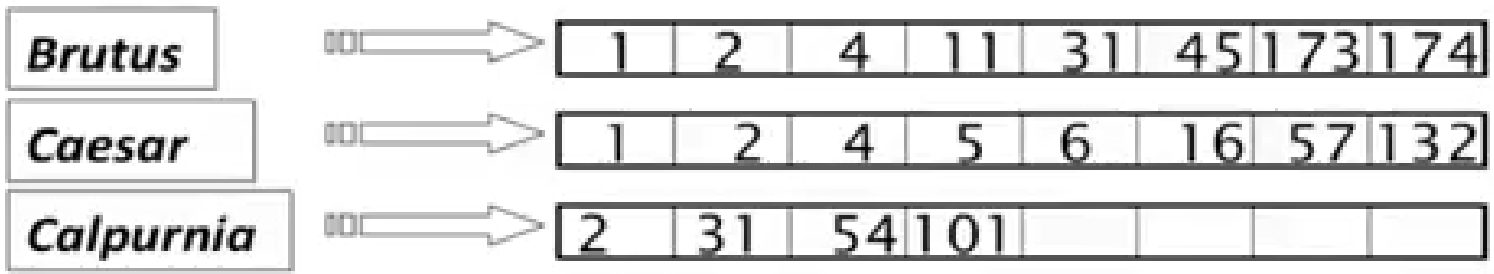
- What type of DS should we use?

Inverted index

Array representation?

- For each term **t**, we must store a list of all documents that contain that term **t**.
  - Identify each doc by **docID**, a document serial number.

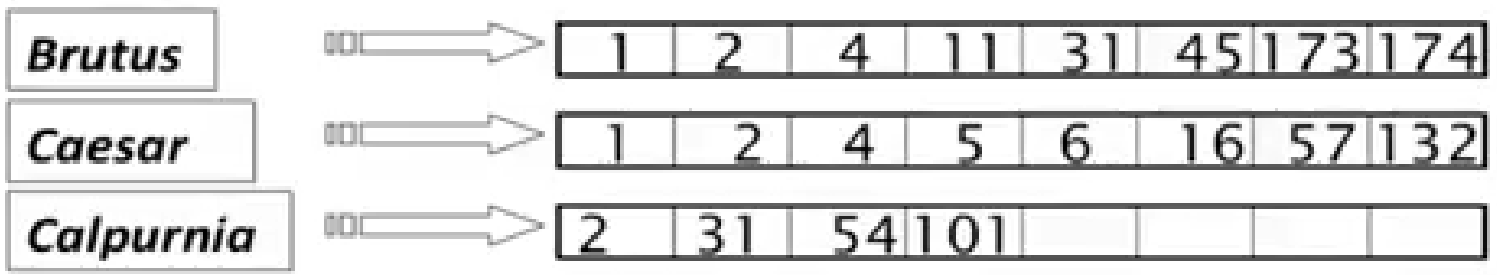
- What type of DS should we use?
  - Can we used fixed-size arrays for this?



Inverted index

Array representation?

- What type of DS should we use?
  - Can we used fixed-size arrays for this?



Inefficient

1 - Some word will appear in many documents other will appear only in few documents

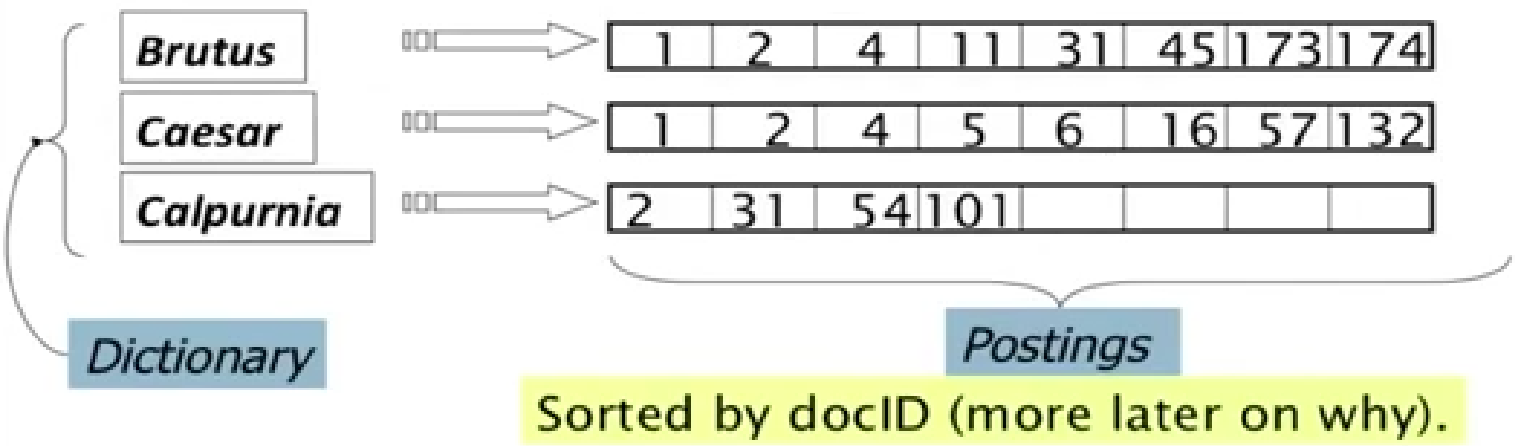
2 - What if the word Caesar is added to document 14?

we will have difficulties in adjusting array size (Problem of updating).

Inverted index

What's the solution?

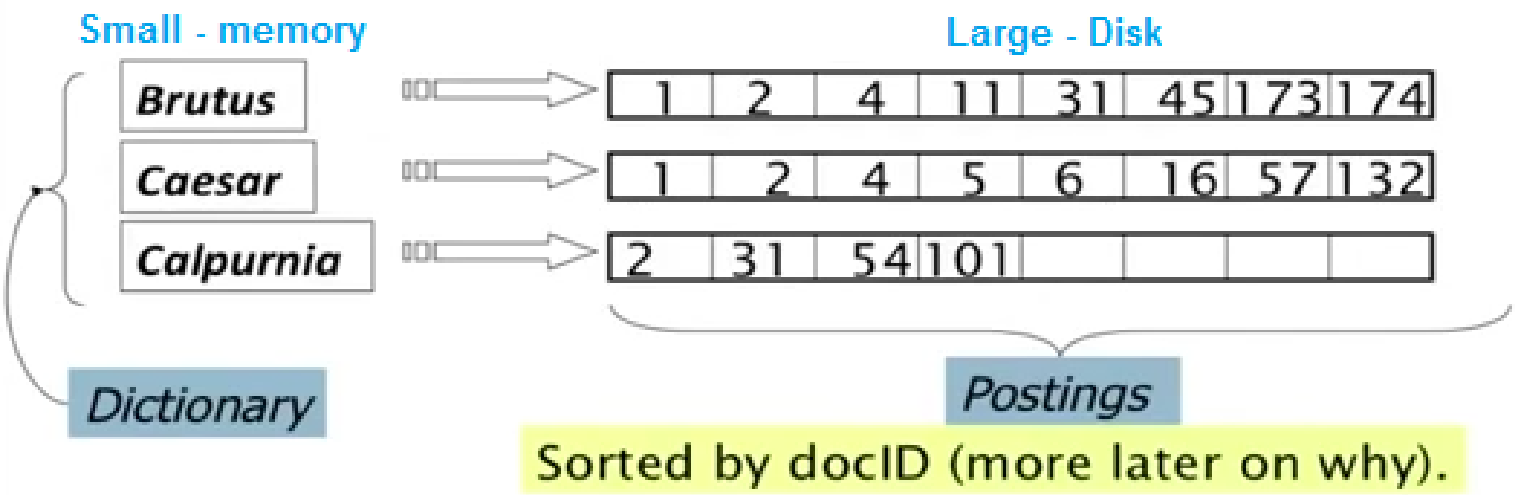
- We need variable-size Postings lists
  - On disk, a continuous run of postings is normal and best.
  - In memory, can use linked lists or variable length arrrays.
    - Or any with some tradeoffs in size/ease of insertion.



Inverted index

What's the solution?

- 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 arrrays.
    - Or any with some tradeoffs in size/ease of insertion.

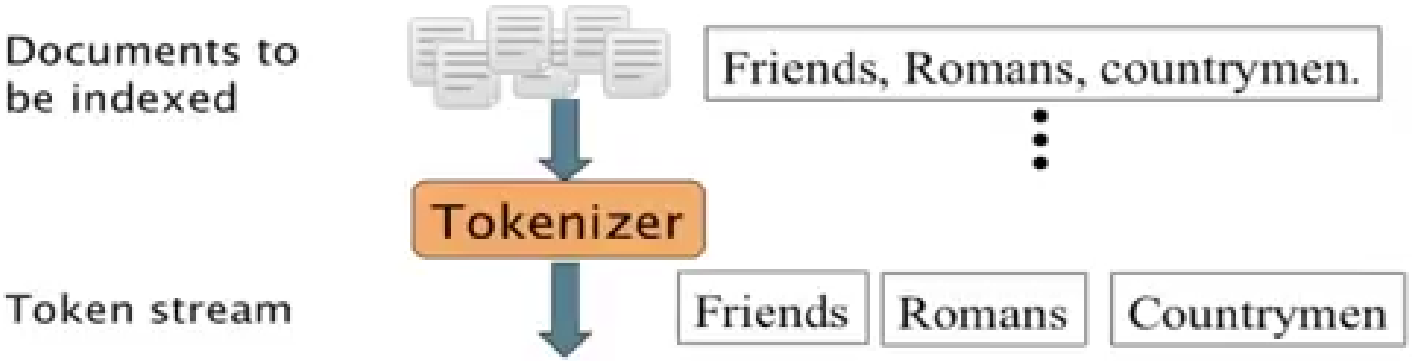


Inverted index construction

What's the essential step of building inverted index

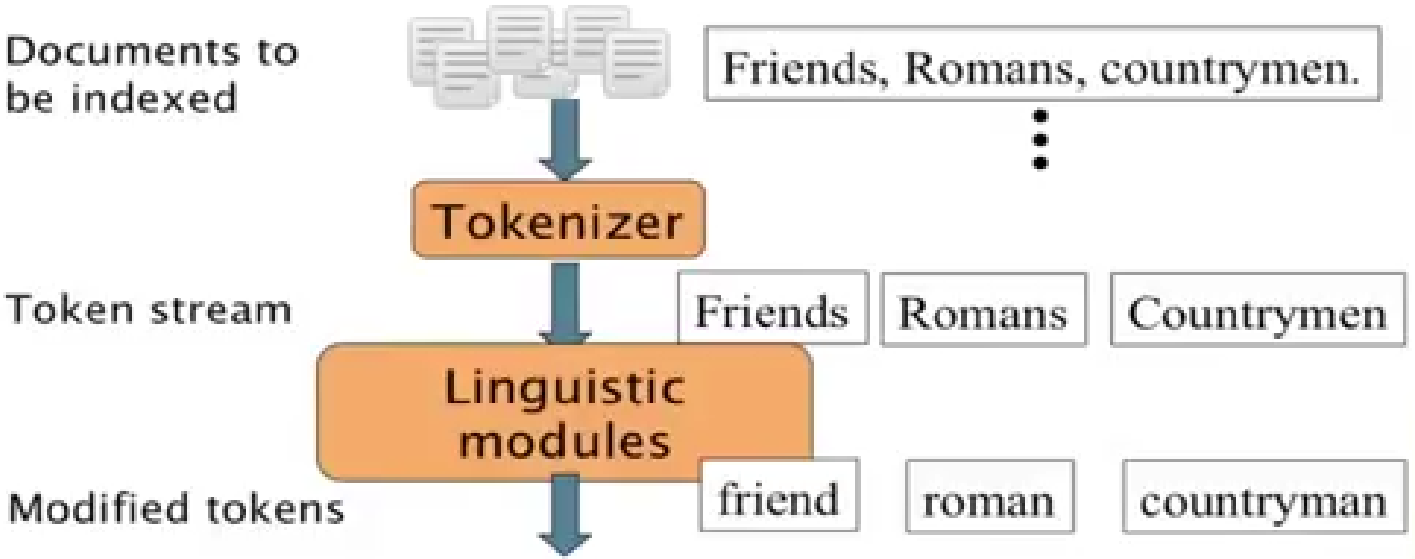
Inverted index construction

What's the essential step of building inverted index



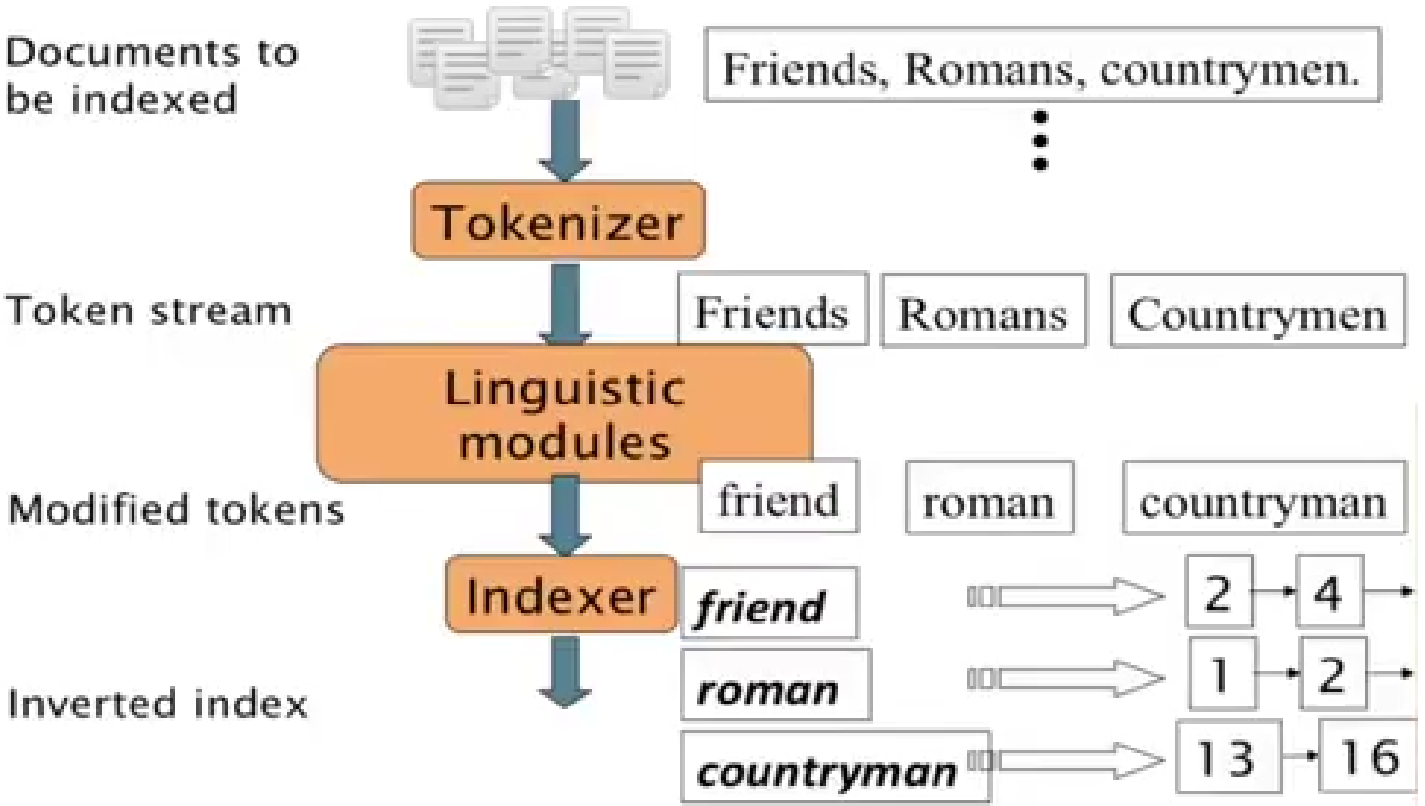
Inverted index construction

What's the essential step of building inverted index



Inverted index construction

What's the essential step of building inverted index



Indexer steps

Step 1 : Token sequence

- 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:

Indexer steps

Step 1 : Token sequence

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

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

Step 2 : Sort - (Core indexing step)

- Sort by terms.

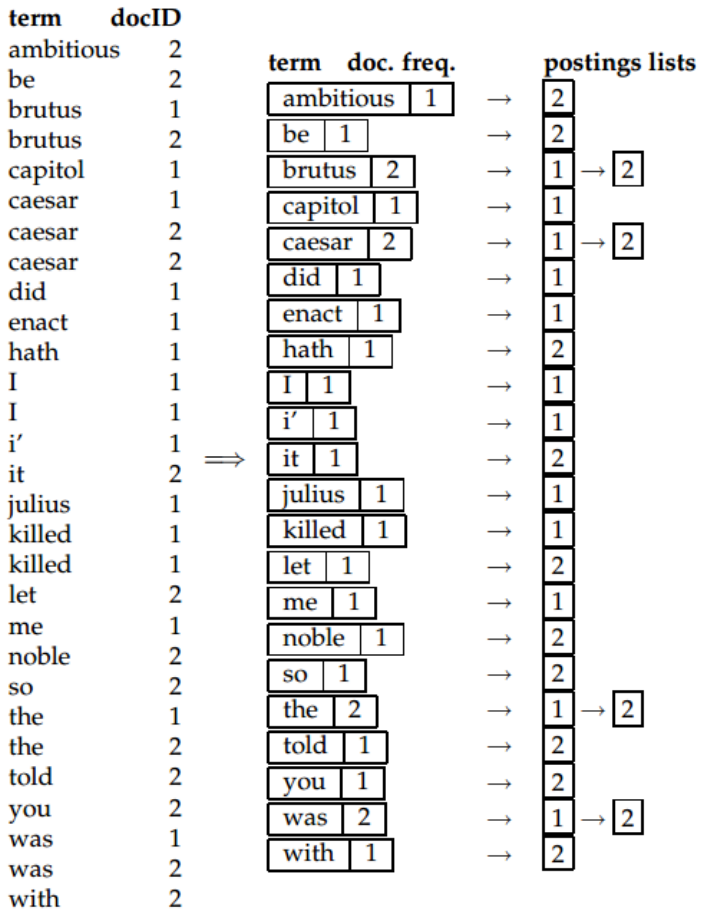
- and then by docID.

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

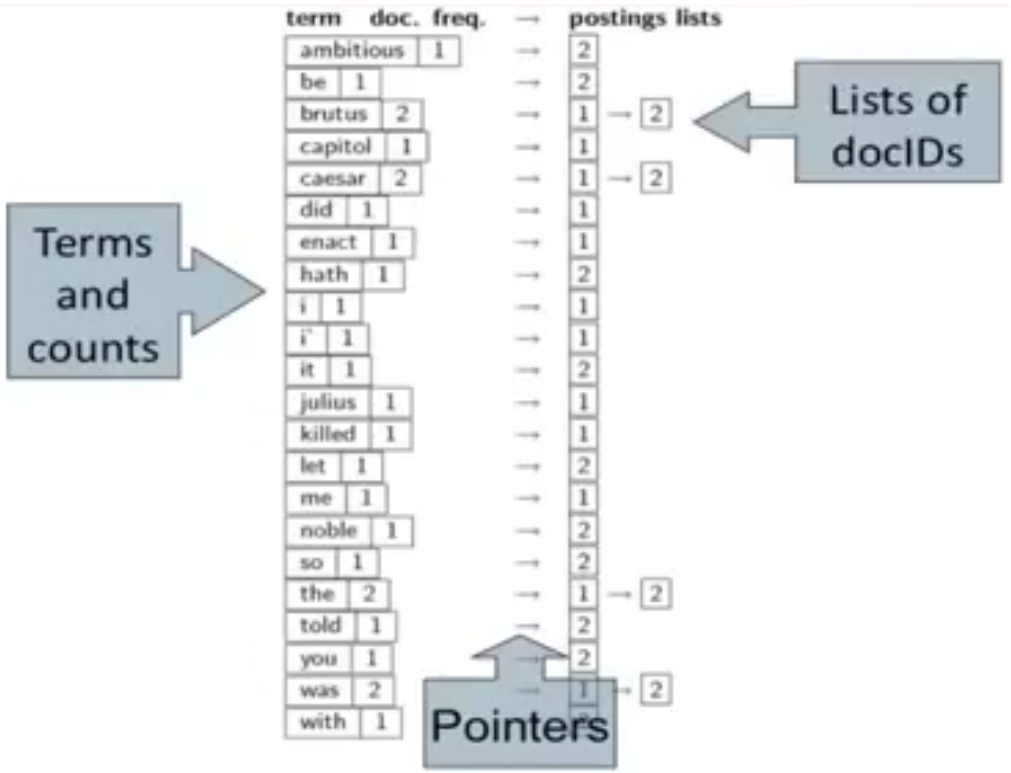
Indexer steps

Step 3 : Dictionary and Postings

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



Where de we pay in storage?



Building efficient IR system

IR system implementation

1. How can we make the indexer as efficient as possible for retrieval.
2. How can we minimize the storage in both sides (Memory - Dictionary, and Disk - Postings).

Query processing with inverted index

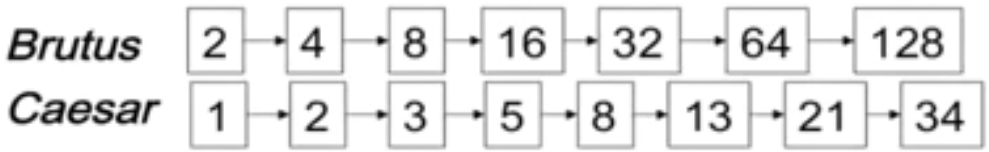
After the index we just built

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

Query processing with inverted index

Consider processing the query Brutus AND Caesar

1. Locate **Brutus** in the Dictionary;
   - Retrieve its postings.
2. Locate **Caesar** in the Dictionary;
   - Retrieve its postings.
3. "Merge" the two postings (intersect the document sets).

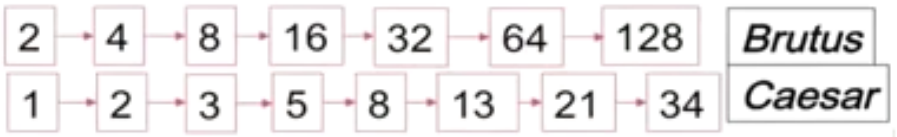


Merge is a misleading word, because normally its refer to put a two sets together (more precisely union operation) but in our case it is not.
Why?
- The term MERGE uses in both cases, The MERGE Algorithms family refer to algorithms that applies different kinds of boolean operations on two sets of sorting list.

Query processing with inverted index

The Merge

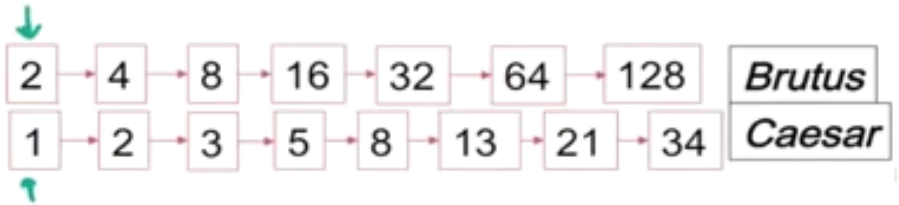
1. Walk through the two postings simultaneously, in time linear in the total number of postings entries.



Query processing with inverted index

The Merge

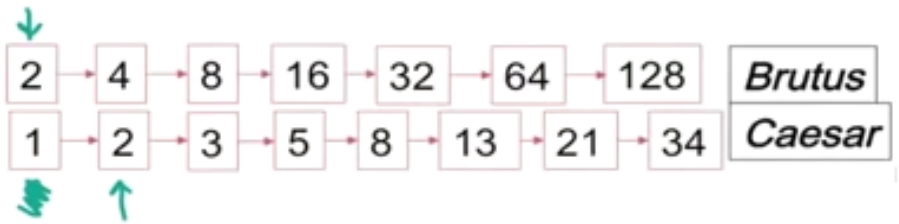
1. Walk through the two postings simultaneously, in time linear in the total number of postings entries.



Query processing with inverted index

The Merge

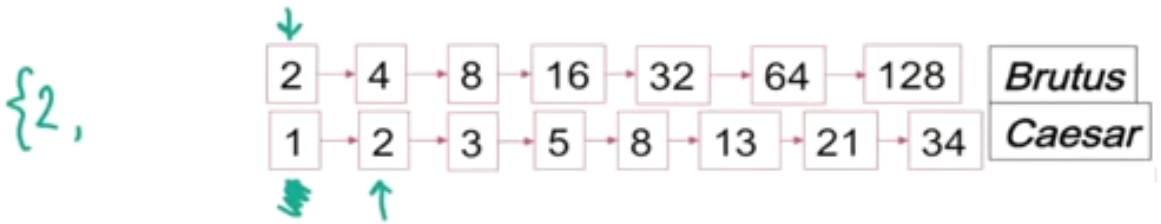
1. Walk through the two postings simultaneously, in time linear in the total number of postings entries.



Query processing with inverted index

The Merge

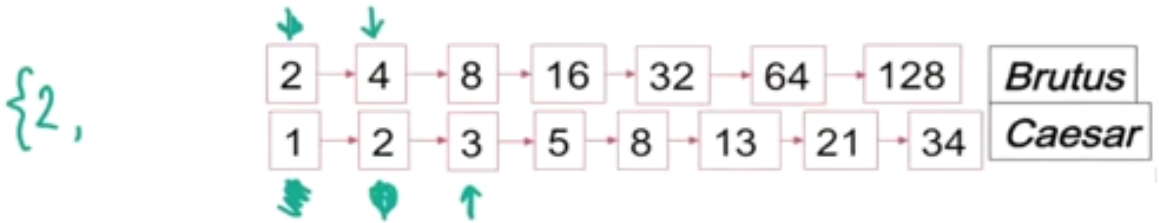
1. Walk through the two postings simultaneously, in time linear in the total number of postings entries.



Query processing with inverted index

The Merge

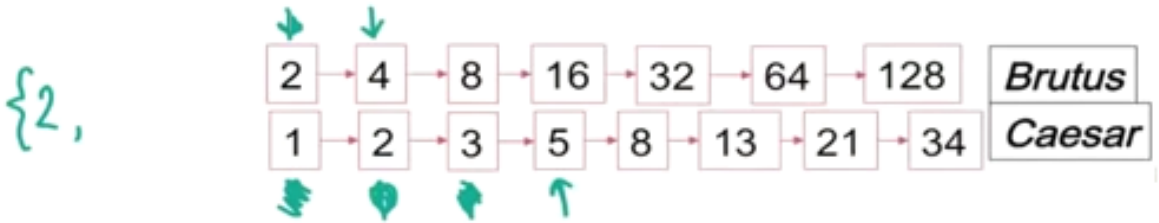
1. Walk through the two postings simultaneously, in time linear in the total number of postings entries.



Query processing with inverted index

The Merge

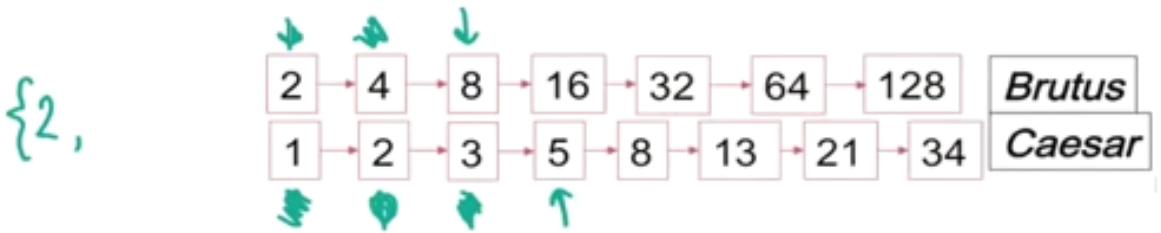
1. Walk through the two postings simultaneously, in time linear in the total number of postings entries.



Query processing with inverted index

The Merge

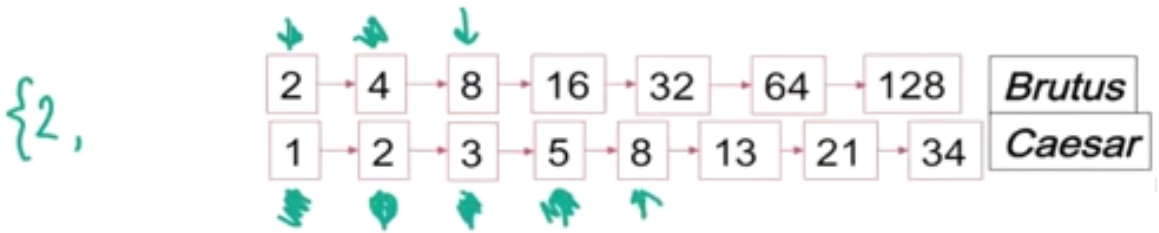
1. Walk through the two postings simultaneously, in time linear in the total number of postings entries.



Query processing with inverted index

The Merge

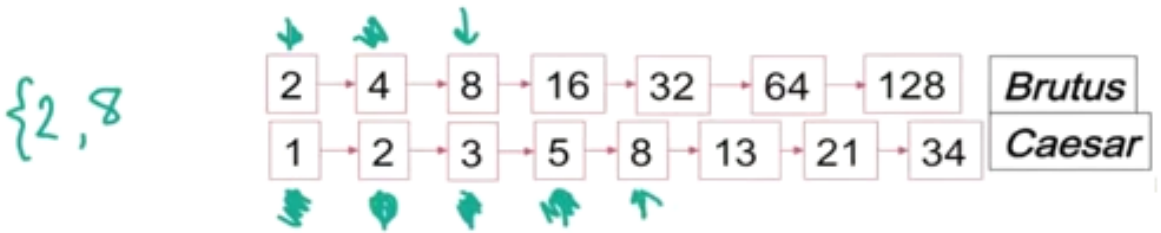
1. Walk through the two postings simultaneously, in time linear in the total number of postings entries.



Query processing with inverted index

The Merge

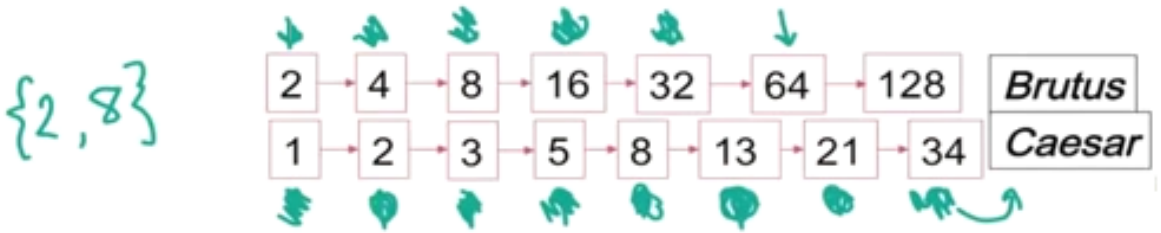
1. Walk through the two postings simultaneously, in time linear in the total number of postings entries.



Query processing with inverted index

The Merge

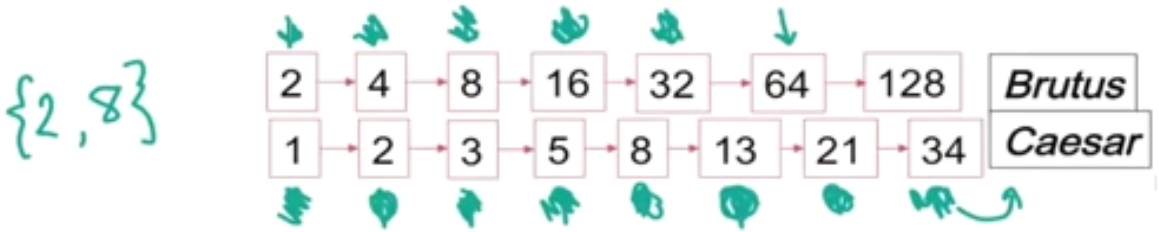
1. Walk through the two postings simultaneously, in time linear in the total number of postings entries.



Query processing with inverted index

The Merge

1. 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

Query processing with inverted index

The algorithm of The Merge

```
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
```

Phrase queries

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

Biword indexes

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 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

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

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.>

processing a phrase query

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

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

Positional index size

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

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
1000	1	1
100,000	1	100