BITS Pilani
K K Birla Goa Campus

innovate achieve lead

2. Boolean Retrieval

Information Retrieval

Some slides are taken from Introduction to Information Retrieval course by CD Manning, Stanford NLP Group

Outline

❖Evaluation of IR System

Content

1. Term Vocabulary
2. Inverted Index
3. Boolean Retrieval
4. Faster Posting Lists
5. Extended Boolean Retrieval
6. Phrase Queries and Positional Postings

Evaluation of an IR System

Evaluation Metrics

	Retrieved	**Not Retrieved**
Relevant	Relevant Retrieved (tp)	Relevant Rejected (fn)
Not Relevant	Irrelevant Retrieved (fp)	Irrelevant Rejected (tn)

$$\boldsymbol{Precision} = \frac{Relevant\ Retrieved}{Retrieved}$$

$$\boldsymbol{Recall} = \frac{Relevant\ Retrieved}{Relevant}$$

How good are the retrieved docs?

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

Question 1: An IR system returns 8 relevant documents and 10 irrelevant documents. There are a total of 20 relevant documents in the corpus. What are the precision and recall on this search?

Answer:

Precision= 8/18= 0.44

Recall= 8/20= 0.4

Question 2: For a given query, there are 16 relevant documents in the collection. The precision of the query is 0.40 and the recall is 0.25. How many documents are there in the result set?

Answer:

Recall= $\frac{Relevant}{16} = 0.25$

$\Rightarrow$Relevant= 4

Precision= $\frac{Relevant\ [4]}{Retrieved} = 0.4$

=> Retrieved= 10 (size of the result set)

Question 3: Explain why both *precision* and *recall* metrics are required to evaluate an IR system and not just one metric individually?

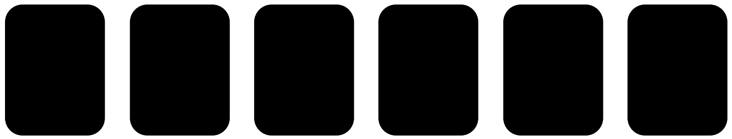
Collection: 50 documents= 15 Relevant + 35 Irrelevant

- **High Recall and Low Precision:**
  - All documents are retrieved irrespective of the query.

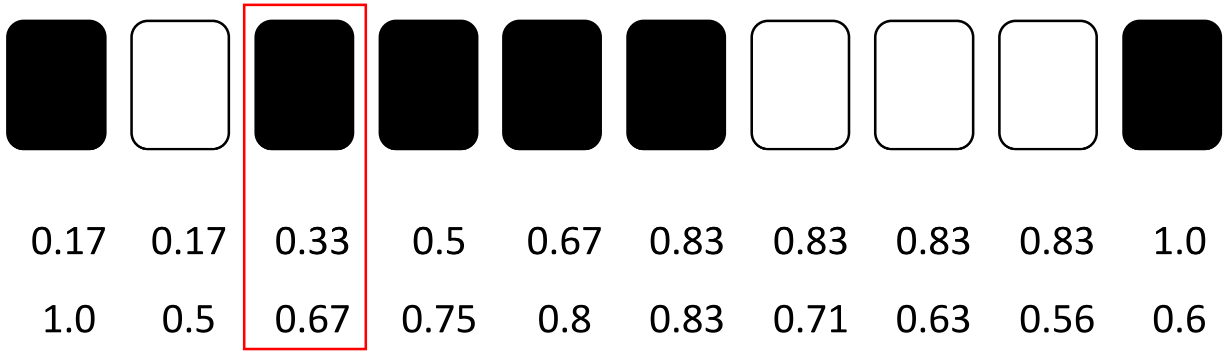
What we retrieved and what we missed.

- **High Precision and Low Recall:**
  - Only 1 document (which is relevant) is retrieved irrespective of the query
  - $Recall = \frac{Relevant\ Retrieved}{Relevant} = \frac{1}{15} = 0.06 = 6\%$
  - $Precision = \frac{Relevant\ Retrieved}{Retrieved} = \frac{1}{1} = 1 = 100\%$

Evaluation Metrics- Ranking



= Relevant documents in collection



Recall	0.17	0.17	0.33	0.5	0.67	0.83	0.83	0.83	0.83	1.0
Precision	1.0	0.5	0.67	0.75	0.8	0.83	0.71	0.63	0.56	0.6

Overall Precision
Overall Recall
Precision and Recall for top K documents

F-measure

- Harmonic mean of Precision and Recall
- $Fscore = \frac{2PR}{P+R}$

R= 0.33

P= 0.67

$$Fscore = \frac{2 * 0.67 * 0.33}{0.67 + 0.33} = \frac{0.4422}{1} = 0.44$$

1. Term Vocabulary

- Tokenization
- Stopwords
- Normalization
- Lemmatization
- Stemming
- Thesaurus

1.1 Token and Terms

- Token is a particular instance of sequence of characters. Example:
  - Friends, and romans, and countrymen forming this one sentence.
  - Tokens:
    - Friends
    - And
    - Romans
    - And
    - Countrymen
    - Forming
    - this
    - One
    - Sentence

- Term is a word in the dictionary.
  - Both tokens ***and*** will map with the same term ***and*** in the dictionary
  - What is being stored in the dictionary.

- Friend is a term
- Friends is a token

Issues in Tokenization

- **Finland's capital:**
  - Finland? Finlands? Finland's?
- **Hyphenated Sequence:**
  - Hewlett-Packard: Hewlett and Packard as two tokens?
  - State-of-the-art technology
  - Co-education
  - Lowercase, lower-case, lower case?
    - Query exapansion
- **White spaces:**
  - San Francisco: one token or two?

Issues in Tokenization

- **Numbers**
  - 3/12/91
    - India: 3rd December
    - US: 12th March
    - Mar. 12, 1991
      - white space?
      - Other formats need to be preserved.
  - B-52/ F-16- name of an aircraft or bomb
  - SSN, PGP keys
  - phone numbers
    - (832) 123-4567
    - Often have embedded spaces
    - Older IR systems may not index numbers
      - But often very useful- things like looking up error codes/stacktraces on the web
      - Can use n-grams
    - Will often index *meta-data* separately
      - Creation date, format, etc.

Issues in Tokenization

- **Language issues:**
  - French
    - L’ensemble: one token or two
      - L? L’? Le?
      - Want *l’ensemble* to match with *un ensemble*
      - Until at least 2003, it didn’t work on Google.
  - German noun compounds are not segmented
    - “Life insurance company employee”
    - Lebensversicherungsgesellschaftsangestellter
    - German retrieval systems benefit greatly from a compound splitter module
      - Can give a 15% performance boost for German

Issues in Tokenization

- **Language issues:**
  - Chinese and Japanese have no spaces between words

汶川地震灾区首批自建永久性农房建成入住

  - Not always guarantee a unique tokenization
  - Further complicated in Japanese, with multiple alphabets intermingled

私はスポーツが好きです

  - Katakana, Hiragana, Kanji, romaji while end user can express query entirely in hiragana.

Issues in Tokenization

- **Language issues:**
  - In Korean language, insertion of spaces in flexible.

    컴퓨터 게임

  - Dates/a
  - January 17, 2018: 2018 년 1 월 17 일
  - Number 17: seventeen: 십칠

Issues in Tokenization

- **Language issues:**
  - Arabic (or Hebrew) is written right to left but with certain items like numbers written left to right
  - Words are separated but letters forms within a word form complex ligatures
  - “Algeria achieved its independence in 1962 after 132 years of French occupation”
  - حققت الجزائر استقلالها في عام 1962 بعد 132 عاما من الاحتلال الفرنسي
  - With Unicode, the surface presentation is complex but the stored form is straightforward

1.2 Stop words

- With a stop list, you exclude from the dictionary entirely the commonest words that appear in almost all the documents.
  - They have little semantic content: the, a, an, and, to, be
  - There are a lot of them: ~30% of postings
- But the trend is away from doing this:
  - Good compression techniques means the space for including stop words in a system is very small
  - You need them for:
    - Phrase queries: “king of Denmark”
    - Various song titles: “let it be, to be or not to be”
    - Relational queries: “flights to London”

1.3 Normalization to Terms

- We need to "normalize" words in indexed text as well as query words into the same form
    - we want to match U.S.A and USA
    - Automation, automate, automatic
- Results is terms: a term is a (normalized) word type, which is an entry in our IR system dictionary
- We most commonly implicitly define equivalence classes of terms by, e.g.,
    - Deleting periods to form a term
        - U.S.A, USA { USA
    - Deleting hyphens to form a term
        - Anti-discriminatory, antidiscriminatory {antidiscriminatory

Normalization: other languages

- Accents
  - French: résumé vs resume
  - Cliché vs cliche
- Umlauts
  - German: Tuebingen vs Tübingen
  - Should be equivalent
- Most important criterion:
  - How are your users like to write their queries for these words?
- Even in languages that standardly have accents, users often may not type them
  - Often best to normalize to a de-accented term
    - Tuebingen, Tübingen, Tubingen { Tubingen

Normalization: other languages

- Normalization of things like date forms
  - 2018 년 1 월 18 일 vs 18/1
  - Japanese use of Kana vs Chinese characters
- Tokenization and normalization may depend on the language and so is intertwined with language detection
  - Morgen will ich in mit

Is this German “mit” or MIT?

- Numbers: 7/10 or 10th July
- Crucial: need to normalize indexed text as well as the query terms into the same form

Case Folding

- Reduce all letters to lower case
  - Query=automobile. A doc has a sentence starting with Automobile should be retrieved.
  - Exception: upper case in mid-sentence?
    - E.g.; General Motors
    - Fed vs fed
    - SAIL vs sail
  - Often best to lower case everything, since users will use lowercase regardless of ‘correct’ capitalization
- Google example:
  - Query C.A.T
  - First few results were for CAT exam and “cat” (animal) not Caterpillar Inc.

Normalization to terms

- An alternative to equivalence classing is to do an asymmetric expansion (query expansion)
- An example of where it may be useful
  - Enter: **window** Search: **window**, **windows**
  - Enter: **windows** Search: **Windows**, **window**, **windows**
  - Enter: **Windows** Search: **Windows**
- Potentially more powerful, but less efficient

1.4 Lemmatization

- A sophisticated NLP technique for normalization
- In English language, words like "am, are, is" are formed from the base word (root or stem form) called "be"
  - car, cars, car's, cars' -> car
  - am, are, is, was -> be
- Reduce inflectional/variant forms to base form
  - the boy's cars are different colors-> the boy car be different color
- Lemmatization implies doing "proper" reduction to dictionary headword form

1.5 Stemming

- Lemmatization requires a fairly sophisticated linguistic analysis of the text
- Stemming is an alternative of lemmatization
- Reduce terms to their "root" before indexing
- Stemming suggests crude affix chopping
  - language dependent
  - e.g. automate(s), automatic, automation all reduced to automat.
  - The final terms may not always be a valid dictionary word

for example compressed and compression are both accepted as equivalent to compress

$\longrightarrow$

for exampl compress and compress ar both accept as equival to compress

Recap

- Precision
- Recall
- F-Score
- Term Vocabulary
    - Tokenization
    - Stopwords
    - Normalization
    - Lemmatization
    - Stemming

Porter’s Stemming

- commonest algorithm for stemming English
  - Results suggest it's at least as good as other stemming options
- 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.
  - Goes through 5 phases of transformation before final term is generated

Phase Example

- phase: a sequence of production rules
  - replacing a string appearing on left side with string appearing on right side
- has a bunch of rules in each phase
  - check if any of these rules has suffice in token
    - sses -> ss
    - ies -> i
    - ational -> ate
    - tional -> tion
- rules sensitive to the measure of words
  - (m>1) EMENT -> (m>1)
    - replacement -> replac
    - cement -> cement

Other Stemmer

- Other stemmer exist, e.g. Lovins stemmer
  - http://snowball.tartarus.org/algorithms/lovins/stemmer.html
  - Single-pass, longest suffix removal (about 250 rules)

- Do stemming and other normalization help?
  - English: very mixed results. Helps recall but harms precision
    - operative (dentistry) -> oper
    - operational (research) -> oper
    - operating (systems) -> oper

Language Specificity

- many of the above features embody transformations that are
    - language specific and
    - often application specific
- these are plug-in addenda to the indexing
- both open source and commercial plug-ins are available for handling these

1.6 Thesaurus

- Do we handle synonyms and homonyms?
  - e.g. by hand-constructed equivalence classes
    - **car=automobile** **color=colour**
  - we can rewrite to form equivalence-class terms
    - when the document contains automobile, index it under **car-automobile** (and vice-versa)
  - or we can use query expansion
    - when the query contains **automobile**, look under **car** as well

Challenges

- Parsing a document
  - Format? Language? Character set? Unit document?
  - Complete book is a one document?
  - Each section or paragraph is a document?
  - Every sentence is a separate document?

2. Inverted Index

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

Term-document incidence matrices

	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

- 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**

Limitations?

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

Bigger collections

- Consider *N* = 1 million documents, each with about 1000 words (10^9 words).
- Average 6 bytes/word including spaces and punctuation
  - 6GB of data in the documents (1GB= 10^9 bytes).
- Say there are *M* = 500K *distinct* terms among these.

Can’t build the matrix

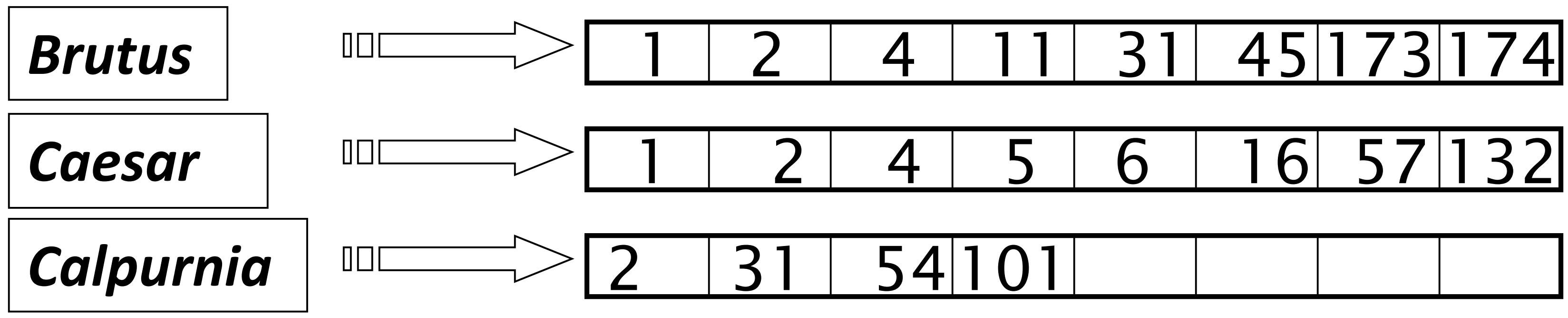
- 500K x 1M matrix ($0.5*10^{12}$) has half-a-trillion 0’s and 1’s.
- But it has no more than one billion 1’s.
  - matrix is extremely sparse.

Inverted Index

- By far, the most commonly and frequently used data structure in modern IR systems.
- **INVERT** index?
- Earlier, a 2D metrics of terms that are present in a document- **sparsity**
- Now, for each term *t*, store a list of documents that contain *t*.

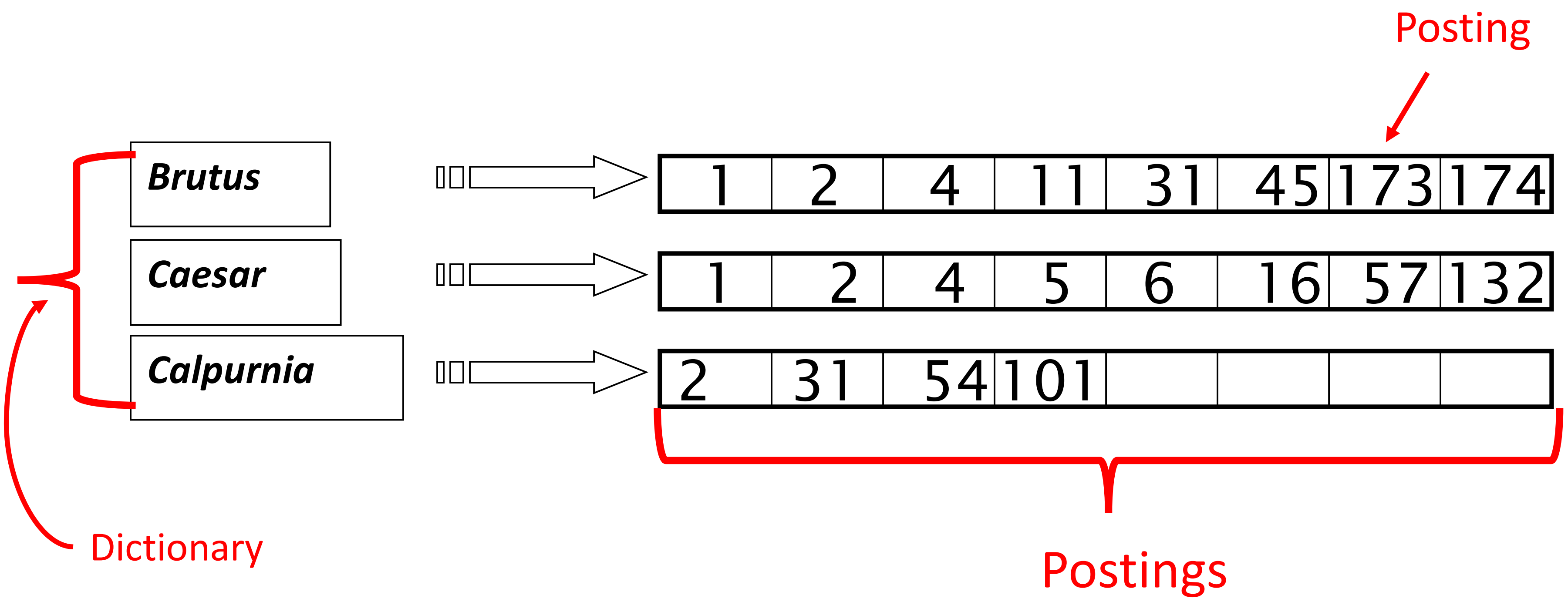
Inverted index

- Assign a unique id to each document. Say, Doc1, Doc2, Doc3
- Sort the documents in a specific (say, ascending) fashion.



Inverted index

- We need variable-size postings lists



Sorted by docID

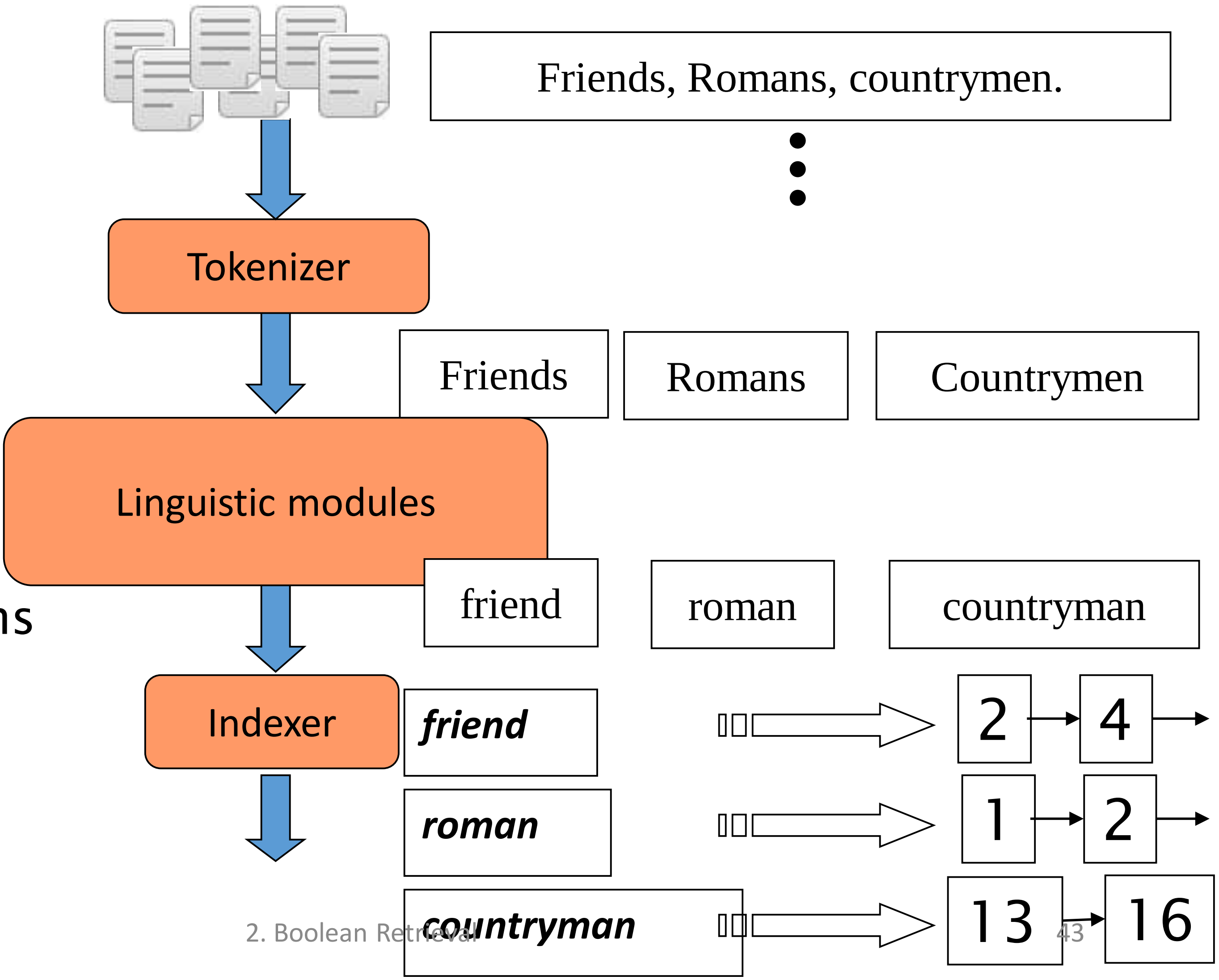
Inverted index construction

Documents to be indexed

Token stream

Modified tokens

Inverted index



Indexer steps: Token sequence

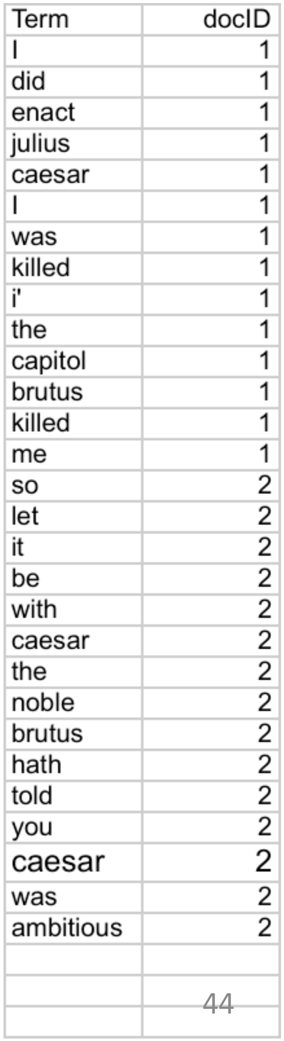
- Sequence of (Modified token, Document ID) pairs.

Doc 1

I did enact Julius Caesar I was killed i’ the Capitol; Brutus killed me.

Doc 2

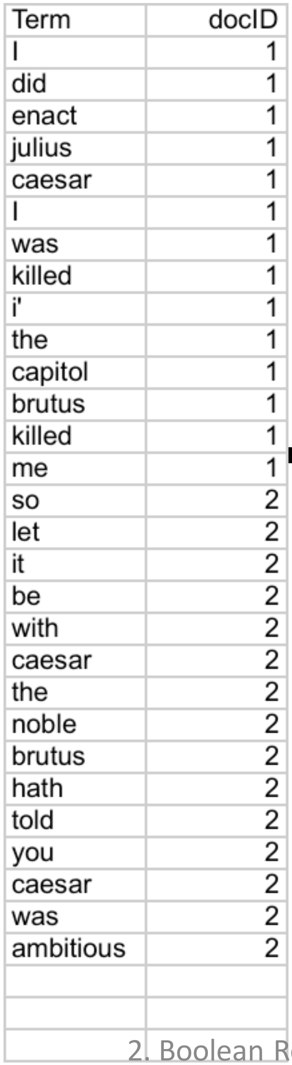
So let it be with Caesar. The noble Brutus hath told you Caesar was ambitious



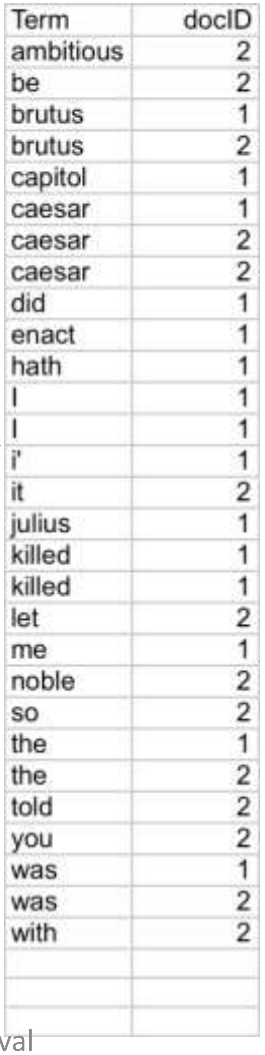
Term	docID
I	1
did	1
enact	1
julius	1
caesar	1
I	1
was	1
killed	1
i'	1
the	1
capitol	1
brutus	1
killed	1
me	1
so	2
let	2
it	2
be	2
with	2
caesar	2
the	2
noble	2
brutus	2
hath	2
told	2
you	2
caesar	2
was	2
ambitious	2

Indexer steps: Sort

- Sort by terms
  - And then docID



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

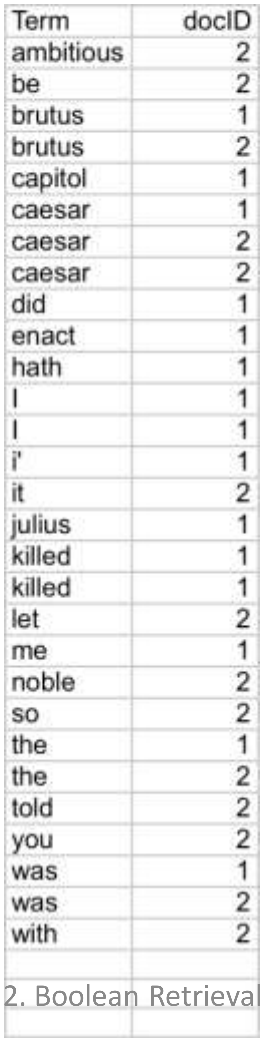


Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2

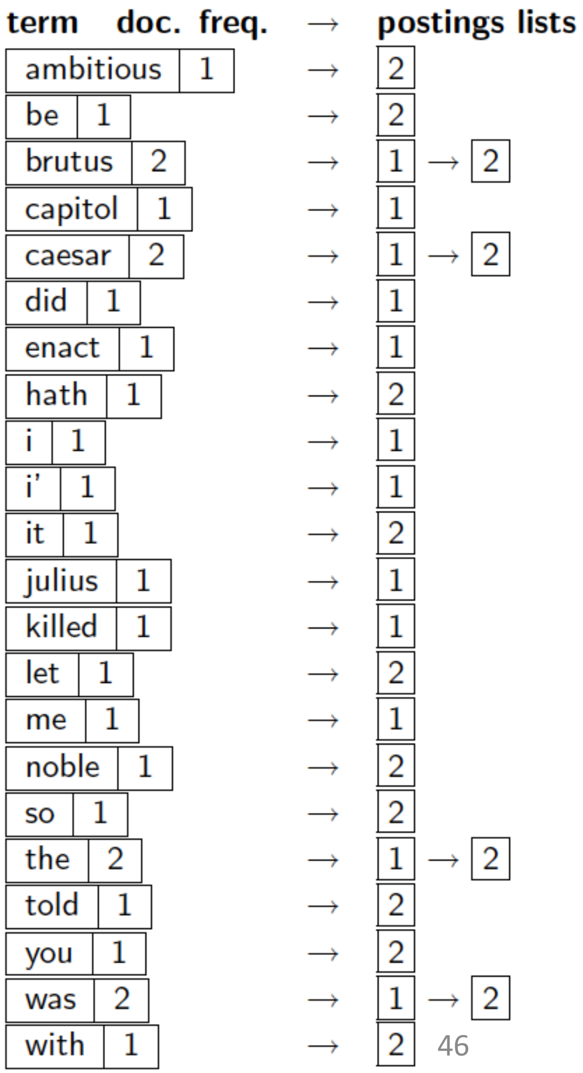
Indexer steps: Dictionary & Postings

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

Why frequency?
Will discuss later.

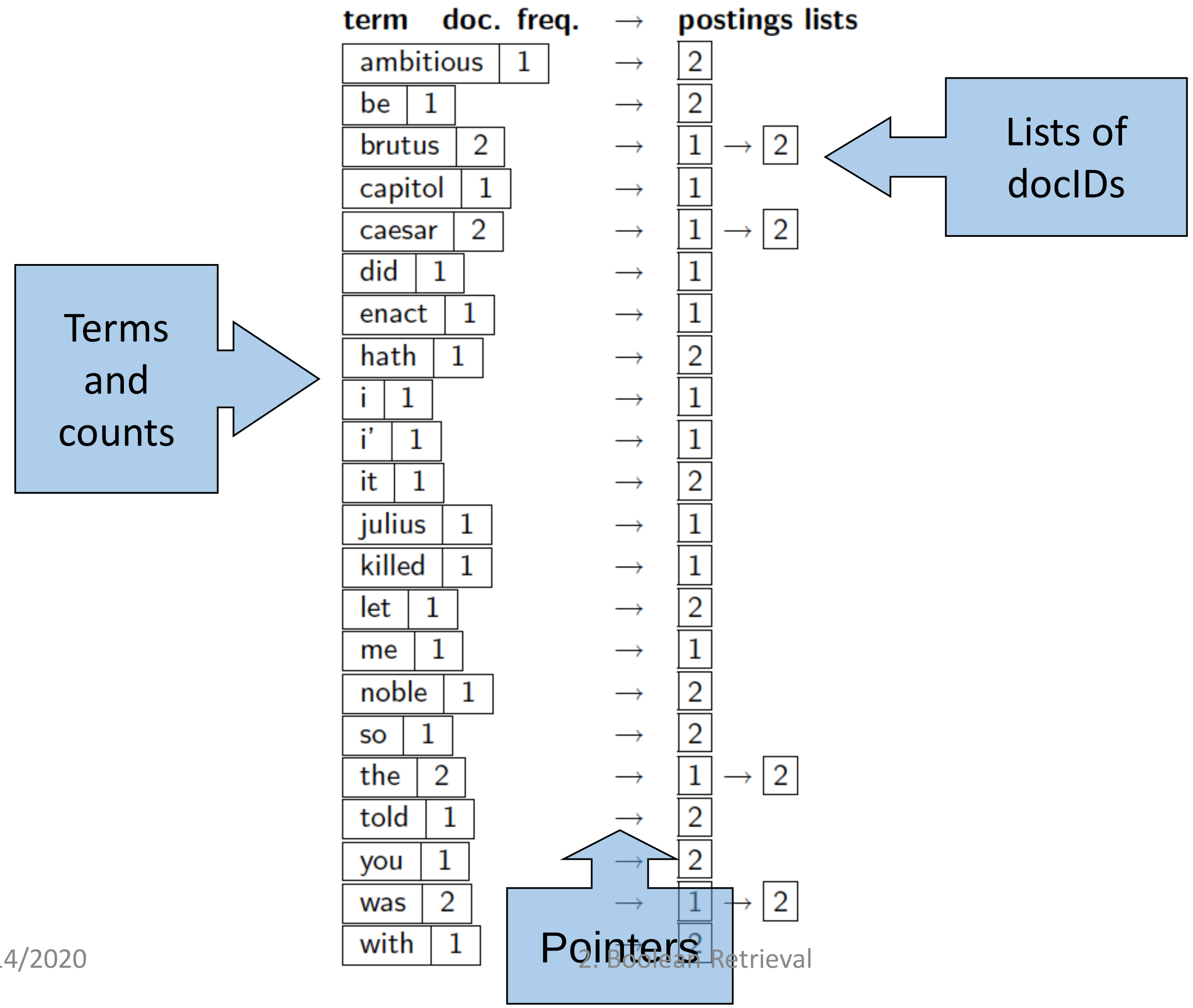


Term	docID
ambitious	2
be	2
brutus	1
brutus	2
capitol	1
caesar	1
caesar	2
caesar	2
did	1
enact	1
hath	1
I	1
I	1
i'	1
it	2
julius	1
killed	1
killed	1
let	2
me	1
noble	2
so	2
the	1
the	2
told	2
you	2
was	1
was	2
with	2



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

Where do we pay in storage?



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

3. The Boolean Retrieval Model

Boolean Model

- Simple model based on set theory and Boolean algebra
  - Documents are sets of terms
  - Queries are Boolean expressions on terms
- Historically the most common model
- Queries specified as boolean expressions
  - Precise semantics
  - Neat formalism
- Terms are either present or absent. Thus, $w_{ij}\ \varepsilon\ \{1,0\}$
- There are three connectives used: *and, or, not*

Boolean Model

- **D: set of words (indexing terms) present in a document**
  - each term is either present (1) or absent (0)
- **Q: A Boolean expression**
  - terms are index terms
  - operators are AND, OR, and NOT
- **F: Boolean algebra over sets of terms and sets of documents**
- **R:** a document is predicted as relevant to a query expression if it *satisfies the query expression*
- Example: ***((text $\vee$ information) $\wedge$ retrieval $\wedge$ $\neg$ theory)***
- Each query term specifies a set of documents containing the term
- AND ($\wedge$): the intersection of two sets
- OR ($\vee$): the union of two sets
- NOT ($\neg$): set inverse, or really set difference

Boolean Queries: **Exact match**

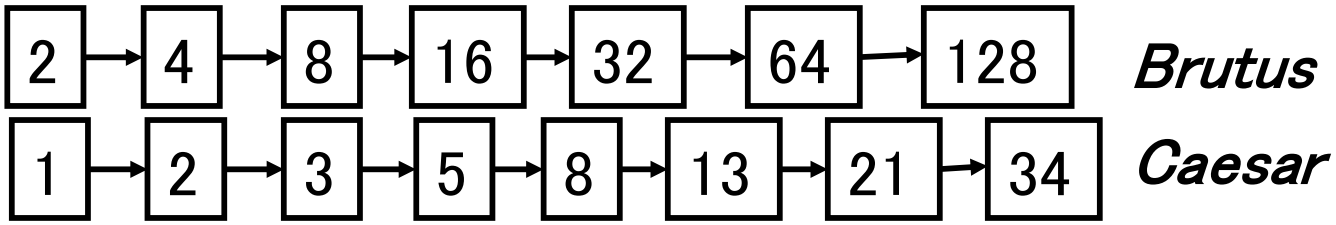
- The Boolean retrieval model is being able to ask a query that is a Boolean expression:
  - Boolean Queries are queries using *AND, OR* and *NOT* to join query terms
    - Views each document as a set of words
    - Is precise: document matches condition or not.
  - Perhaps the simplest model to build 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

Query processing: *x* AND *y*

- 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):



$\boxed{2}$ $\boxed{8}$

Documents to be retrieved.

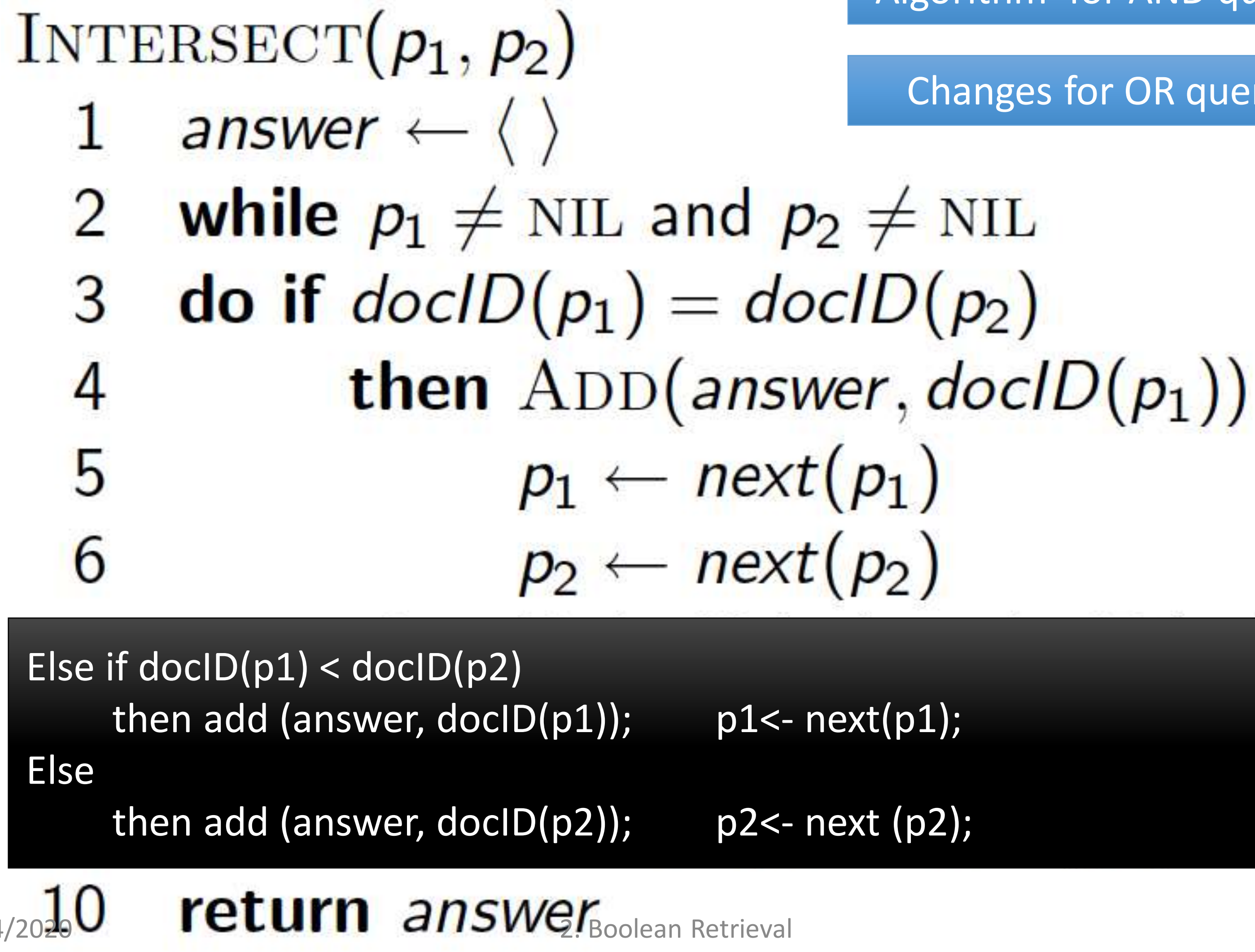
Intersecting two postings lists (a “merge” algorithm)

```
INTERSECT($p_1$, $p_2$)
answer ← ⟨ ⟩
while $p_1 \neq$ NIL and $p_2 \neq$ NIL
do if $docID(p_1) = docID(p_2)$
      then ADD(answer, $docID(p_1)$)
           $p_1 \leftarrow next(p_1)$
           $p_2 \leftarrow next(p_2)$
      else if $docID(p_1) < docID(p_2)$
              then $p_1 \leftarrow next(p_1)$
              else $p_2 \leftarrow next(p_2)$
return answer
```

Query Processing: *x* OR *y*

Algorithm for AND query.

Changes for OR query.



```
INTERSECT($p_1$, $p_2$)
1   $answer \leftarrow \langle\, \rangle$
2   while $p_1 \neq$ NIL and $p_2 \neq$ NIL
3   do if $docID(p_1) = docID(p_2)$
4         then ADD($answer$, $docID(p_1)$)
5              $p_1 \leftarrow next(p_1)$
6              $p_2 \leftarrow next(p_2)$
```

```
Else if docID(p1) < docID(p2)
        then add (answer, docID(p1));        p1<- next(p1);
Else
        then add (answer, docID(p2));        p2<- next (p2);
```

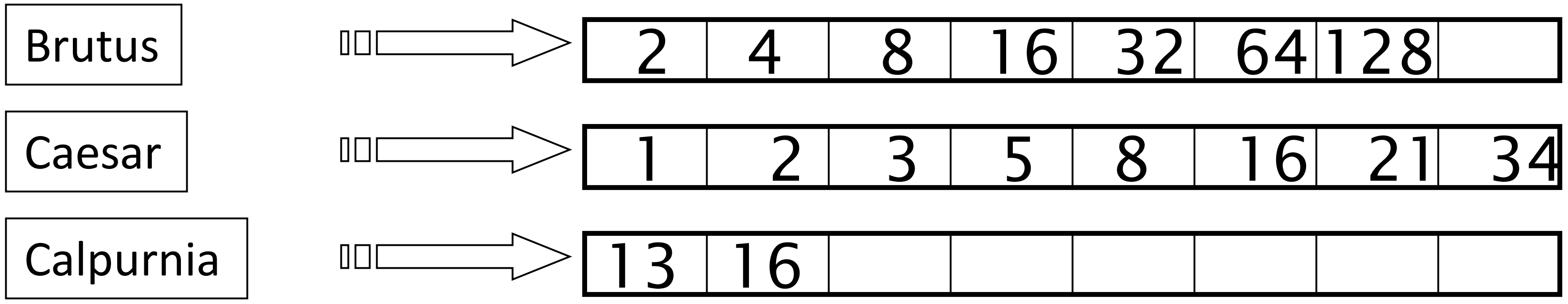
```
10  return $answer$
```

READING TASK- 1

- Query Processing algorithm:
  - *x* AND (NOT *y*)

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.



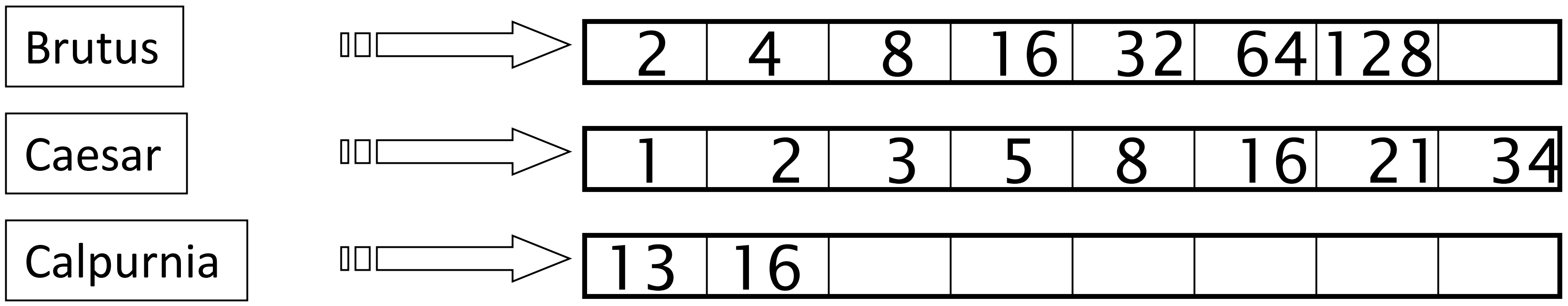
Query: ***Brutus*** *AND* ***Calpurnia*** *AND* ***Caesar***

56

Query optimization example

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

This is why we kept document freq. in dictionary



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

Exercise- OR query

- 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.

Advantages

- Clean Formalism
- Easy to implement
- Intuitive concept
- Still it is dominant model for document database systems

Limitations

- Retrieval based on binary decision criteria with no notion of partial matching
- No ranking of the documents is provided (absence of a grading scale)
- Information need has to be translated into a Boolean expression which most users find awkward
- The Boolean queries formulated by the users are most often too simplistic
- Frequently returns either too few or too many documents in response to a user query

Exercise

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

4. Faster Posting Lists

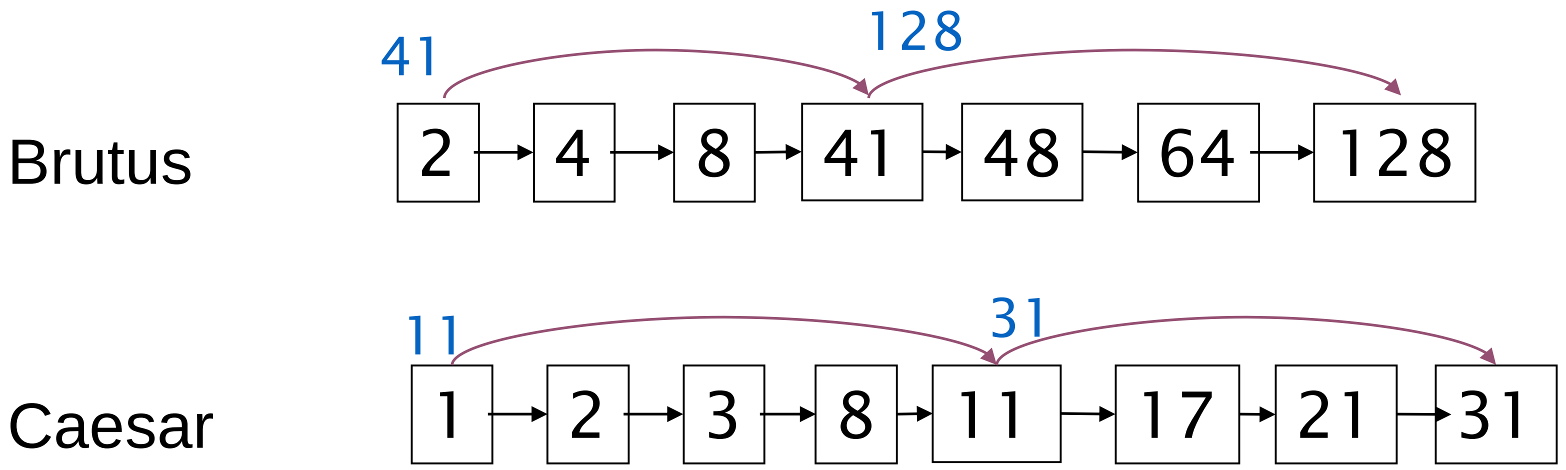
Faster Posting Lists

- Faster postings list intersection via skip pointers
    - ways to increase the efficiency of using postings lists

$$\begin{tikzcd}
\text{Brutus} \arrow[r] & \boxed{1} \arrow[r] & \boxed{2} \arrow[r] & \boxed{4} \arrow[r] & \boxed{11} \arrow[r] & \boxed{31} \arrow[r] & \boxed{45} \arrow[r] & \boxed{173} \arrow[r] & \boxed{174} \\
\text{Calpurnia} \arrow[r] & \boxed{2} \arrow[r] & \boxed{31} \arrow[r] & \boxed{54} \arrow[r] & \boxed{101} \\
\text{Intersection} \arrow[r, Rightarrow] & \boxed{2} \arrow[r] & \boxed{31}
\end{tikzcd}$$

Skip Pointers

- Augmenting postings lists with skip pointers
- Skip pointers are effectively shortcuts that allow us to avoid processing parts of the postings list that will not figure in the search results.
- Where to place skip pointers ?
- How to do efficient merging using skip pointers?



Postings lists intersection with skip pointers

```
IntersectWithSkips(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
            if hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2)) then
                while hasSkip(p1) and (docID(skip(p1)) ≤ docID(p2))
                        do p1 ← skip(p1)
                else p1 ← next(p1)
            else if hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1)) then
                    while hasSkip(p2) and (docID(skip(p2)) ≤ docID(p1))
                        do p2 ← skip(p2)
                else p2 ← next(p2)
return answer
```

Skip Pointers

- Available only for the original postings
- For an intermediate result in a complex query, the call hasskip(p) will always return false
- The presence of skip pointers only helps for AND queries, not for OR queries.

Placing Skips

- Simple heuristic: for postings of length P, use ⍰P evenly-spaced skip pointers.
- This ignores the distribution of query terms.
- Easy if the index is relatively static; harder if P keeps changing because of updates.
- Building effective skip pointers is easy if an index is relatively static
- Difficult if a postings list keeps changing because of updates

5. Phrase Queries and Positional Postings

Phrase Queries

- 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 sentence "The inventor Stanford Ovshinsky never went to university" 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

Handling Phrase Queries

- For this, it no longer suffices to store only
  - <term : docs> entries
- The two approaches to support phrase queries and their combination are
  1. Biword Indexes
  2. Positional Indexes
- A search engine should not only support phrase queries, but implement them efficiently.

5.1 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 phrases are processed by breaking them.
- *“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).
- Call any string of terms of the form NX*N an extended biword.
    - Each such extended biword is now made a term in the dictionary.

Example

- cost overruns on a power plant

  N N X X N N

- Query processing: parse it into N’s and X’s
- Segment query into enhanced biwords
- “cost overruns” AND “overruns power” AND “power plant”
- Better results can be obtained by using more precise part-of-speech patterns that define which extended biwords should be indexed.

Limitations

- False positives
- Index blowup due to bigger dictionary
- Biword indexes are not the standard solution (for all biwords) but can be part of a compound strategy.

5.2 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*: frequency of the term; position1, position2 … ;
  - *doc2*: frequency of the term; position1, position2 … ;
  - etc.>

to, **993427: (**
1, 6: (7, 18, 33, 72, 86, 231);
2, 5: (1, 17, 74, 222, 255);
4, 5: (8, 16, 190, 429, 433);
5, 2: (363, 367);
7, 3: (13, 23, 191); . . .
…
)

be, **178239: (**
1, 2: (17, 25);
4, 5: (17, 191, 291, 430, 434);
5, 3: (14, 19, 101); . . .
…
…
)

Positional index

- For phrase queries, we use a merge algorithm recursively at the document level.
- This requires more than just equality.
- Start with the least frequent term and then work to further restrict the list of possible candidates.
- Need to check that their positions of appearance in the document are compatible with the phrase query being evaluated.
- This requires working out offsets between the words
- Positional Index can also be used for Proximity Searches

Proximity Queries

- ***Information need**: information on the legal theories involved in preventing the disclosure of trade secrets by employees formerly employed by a competing company.*
- ***Query**:* "trade secret" /s disclos! /s prevent /s employe!
- ***Information need**: what are the requirements for disabled people to be able to access a workplace.*
- ***Query:*** disab! /p access! /s work-site work-place (employement /3 place)
- 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