Information Retrieval (CSD510)

Boolean Retrieval

Ayan Das

Classic IR models

- Boolean model
- Vector Space model
- Probabilistic model

Basic concepts (Terminology.)

1. k_i be an index term
2. d_j is a document
3. t - Total number of index terms
4. $K = \{k_1, k_2, \cdot, k_t\}$ - Set of all index terms.
5. w_{ij} weight associated with (k_i, d_j), 0 indicates absence of k_i in d_j.
6. $vec(d_j) = (w_{1j}, w_{2j}, \cdot, w_{tj})$ is the weight vector indicating the weights associated with the index terms in d_j.
7. $g_i(vec(d_j))$ - function returning the weight associated with (k_i, d_j).

Boolean model

- Simple model based on set theory and Boolean algebra.
  - Documents are sets of terms
  - Queries are Boolean expressions on terms
- Queries specified as boolean expressions.
- Terms are either present or absent.$w_{ij} \in \{0, 1\}$.
- There are three connectives used
  - AND ($\wedge$): the intersection of two sets
  - OR ($\vee$): the union of two sets
  - NOT ($\neg$): set inverse, or set difference
- **Document:** A set of words (indexing terms) present in a document
  - each term is either present (1) or absent (0)
- **Query:** A Boolean expression.
  - Effective terms are index terms.
- **Operation:** Boolean algebra over sets of terms and sets of documents.
- **Relevant:** A document is relevant to a query expression if it satisfies the query expression

Boolean Retrieval

- Term-Document Matrix
- Inverted Index

Example: Boolean retrieval

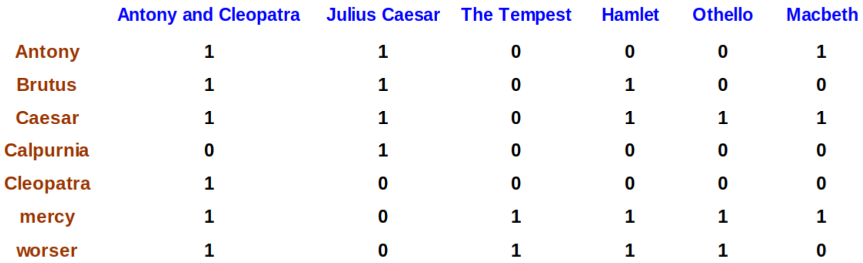
- **Document set:** All plays of Shakespeare.
- **Query:** BRUTUS AND CAESAR AND NOT CALPURNIA
- **Task:** Find all Shakespeare's plays that satisfy the query

A possible solution

- A linear scan of documents (BRUTE FORCE).
  1. **grep** for all plays containing the words *BRUTUS* and *CAESAR*.
  2. From them, strip out all the plays containing the word *CALPURNIA*.
- **Cons**
  1. Slow for large data collection (e.g., the web, which contains billions or trillions of words)
- **A better solution:** Organize and index the documents into better representation to enable more efficient search.

Term-Document Incidence Matrix

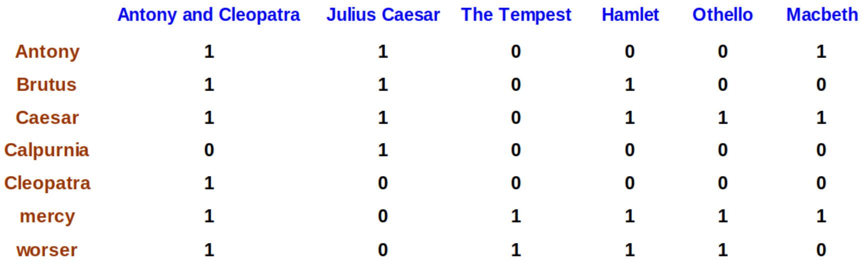
- **Two dimensional:** Terms and documents
- Matrix element (t, d) = 1 if term *t* appears in document *d*



	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

Brutus AND Caesar AND NOT Calpurnia

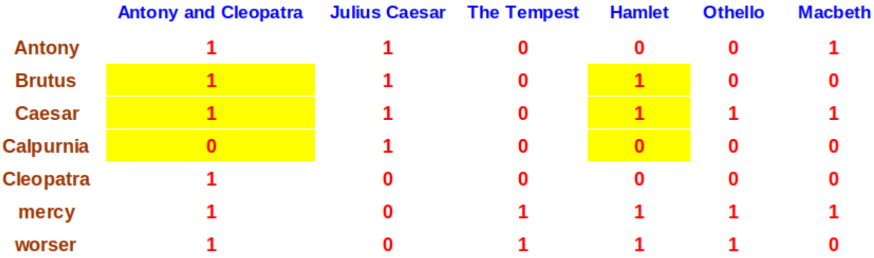
Term-Document Incidence Matrix



	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

Brutus	110100
Caesar	110111
Calpurnia	010000
Brutus AND Caesar	110100
NOT Calpurnia	101111
Brutus AND Caesar AND (NOT Calpurnia)	**100100**

Retrieval result



	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

- The incidence matrices are **usually sparse**.
- Difficult to build for too big Document Corpus.

Bigger collections

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

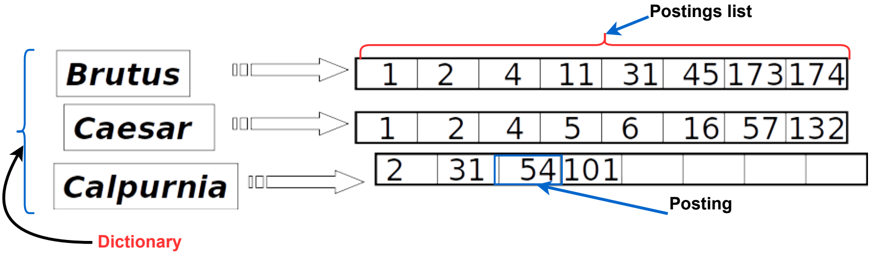
  6GB of data in the documents.
- Say there are M = 500K distinct terms among these.

Can't build the matrix

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

  matrix is extremely sparse.
- What's a better representation?
- Solution is to record only if a term appears in a document.

Inverted Index



Postings list

Brutus → 1 | 2 | 4 | 11 | 31 | 45 | 173 | 174

Caesar → 1 | 2 | 4 | 5 | 6 | 16 | 57 | 132

Calpurnia → 2 | 31 | 54 | 101

Posting

Dictionary

Building inverted index

Preprocessing

1. Collect documents to be indexed
2. Tokenize the text, turning each document into a list of tokens
3. Identify the index terms to form the **vocabulary**
4. Do linguistic pre-processing, producing a list of normalized tokens, which are the indexing terms

Inverted index construction

1. Identify each document by a unique identifier (**docID**).
2. For each term **t** in the vocabulary
   - prepare a list of documents in which the term appears.
   - sort the list on the docIDs.
3. Can be implemented using either singly linked lists or variable length arrays

Inverted index

Consider the following documents

Doc 1: Breakthrough vaccine for Covid

Doc 2: New Covid vaccine

Doc 3: A new approach to vaccination against Covid

Doc 4: New hopes for Covid patients

- **Tokens:** Breakthrough, vaccine, for, Covid, New, A, new, approach, to, vaccination, against, hopes, patients
- **Case normalization:** breakthrough, vaccine, for, covid, a, new, approach, to, vaccination, against, hopes, patients
- **Stopword removal** breakthrough, vaccine, covid, new, approach, vaccination, against, hopes, patients (a, for, to)
- **Stemming:** breakthrough, **vaccin**, covid, new, approach, against, **hope**, **patient**
- **Index terms:** breakthrough, vaccin, covid, new, approach, against, hope, patient

Inverted index

Sort by **docID**

breakthrough	1
vaccin	1
covid	1
new	2
covid	2
vaccin	2
new	3
approach	3
vaccin	3
against	3
covid	3
new	4
hope	4
covid	4
patient	4

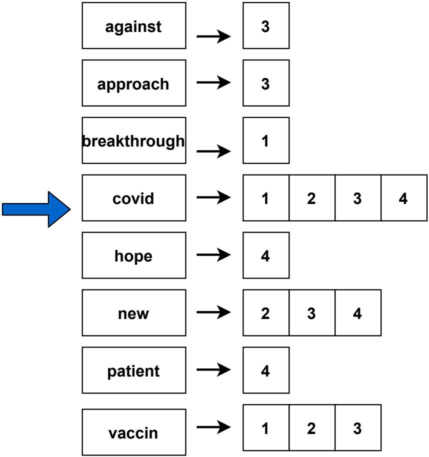
Sort by **terms**

against	3
approach	3
breakthrough	1
covid	1
covid	2
covid	3
covid	4
hope	4
new	2
new	3
new	4
patient	4
vaccin	1
vaccin	2
vaccin	3

Building Inverted Index

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

against	3
approach	3
breakthrough	1
covid	1
covid	2
covid	3
covid	4
hope	4
new	2
new	3
new	4
patient	4
vaccin	1
vaccin	2
vaccin	3



against → 3

approach → 3

breakthrough → 1

covid → 1 2 3 4

hope → 4

new → 2 3 4

patient → 4

vaccin → 1 2 3

Boolean Retrieval

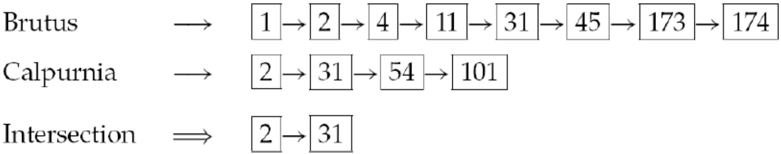
- Processing Boolean queries
- Term vocabulary and postings lists

Practical considerations

- For a practical IR system handling a huge corpus
- Postings lists will be stored on disk.
- Ideally, retrieve (from disk) only those postings lists that are needed to answer a query.

Processing Boolean Queries

- Consider the query: Brutus AND Calpurnia
  1. Locate *Brutus* in the Dictionary
  2. Retrieve its postings
  3. Locate *Calpurnia* in the Dictionary
  4. Retrieve its postings
  5. Intersect the two postings lists



Brutus ⟶ 1 → 2 → 4 → 11 → 31 → 45 → 173 → 174

Calpurnia ⟶ 2 → 31 → 54 → 101

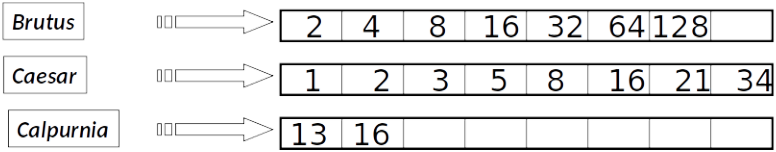
Intersection ⟹ 2 → 31

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

- **Query:** Brutus **AND** Calpurnia **AND** Caesar
- For each of the *n* terms, get its postings, then AND them together.



Brutus → 2 | 4 | 8 | 16 | 32 | 64 | 128

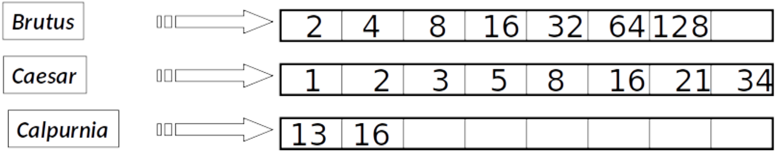
Caesar → 1 | 2 | 3 | 5 | 8 | 16 | 21 | 34

Calpurnia → 13 | 16

```
INTERSECT($p_1, p_2$)
answer $\leftarrow \langle\ \rangle$
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 optimization

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



Brutus	2	4	8	16	32	64	128	
Caesar	1	2	3	5	8	16	21	34
Calpurnia	13	16						

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

- If the list lengths are x and y, the merge takes O(x+y) operations.
- Crucial: postings sorted by docID.

Query processing

- (**wind** OR **fire**) AND (**thunder** OR **lightning**)
- Get doc. frequencies for all terms.
- Estimate the size of each OR by the sum of its document frequencies.
- Process in increasing order of OR sizes.

Query processing

Given the following postings list sizes:

- Recommend a query processing order for the following two queries
  1. (tangerine OR trees) AND (marmalade OR skies) AND (kaleidoscope OR eyes)
  2. (tangerine AND (NOT trees)) AND (NOT marmalade)

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

Query processing

(tangerine OR trees) (363,465)

(marmalade OR skies) (379,571)

(kaleidoscope OR eyes) (300,321)

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

(tangerine AND (NOT trees)) AND (NOT marmalade)

Limitations of Boolean model

- 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
- Binary term weights extremely limited in terms of **expressiveness** and **relation among contextual words**.

Lecture outline

1. Term vocabulary
2. Skip pointers
3. Phrase queries
4. Dictionary structures

Term Vocabulary and Postings List

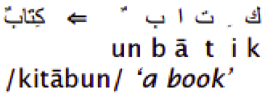
- Pre-processing to form the Term vocabulary
    - Documents
    - Tokenization
    - Indexing

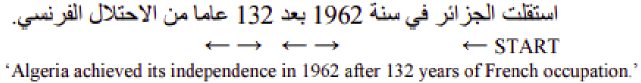
Term Vocabulary and Postings List

- Pre-processing to form the Term vocabulary
  - Documents
  - Tokenization
  - Indexing
- Postings
  - Faster merges: skip lists
  - Positional postings and phrase queries

Document interpretation

- Obtaining the character sequence in a document.
- Choosing a document features
  - We need to deal with format and language of each document.
  - What format is it in? pdf, word, excel, html etc.
  - Language of the document





استقلت الجزائر في سنة 1962 بعد 132 عاما من الاحتلال الفرنسي.
← → ← → ← START
'Algeria achieved its independence in 1962 after 132 years of French occupation.'

Document processing steps for vocabulary generation

- Tokenization
- Stop words
- Normalization
- Stemming and Lemmatization

Tokenization

- **Token:** An instance of character sequence in some particular document that are grouped together as a semantic unit for processing.
- **Type:** A type is the class of all tokens containing the same character sequence.
- **Term:** A term is a type that is included in the IR system's dictionary.
- Tokenization is a way of separating a document into smaller units, called *tokens*, by removing unwanted tokens.

  Example of tokenization

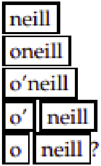
  Input: **"Friends, Romans, Countrymen"**

  Output: Friends, Romans, Countrymen
- Each such token is now a candidate for an index entry, after further processing.

Issues in Tokenization

What are the correct tokens to use?

- Mr. O'Neill thinks that the boys' stories about Chile's capital aren't amusing.



neill

oneill

o'neill

o' neill

o neill ?

- **Hypens**
  - Hewlett-Packard
  - Hewlett and Packard as two tokens?
  - state-of-the-art
  - co-education
  - lowercase, lower-case, lower case
- **White Space**
  - San Francisco: one token or two?
  - red herring: one token or two?

Issues in Tokenization

- Different character sequences
  - email addresses (jblack@mail.yahoo.com)
  - Web URLs (http://stuff.big.com/new/specials.html)
  - numeric IP addresses (142.32.48.231)
  - package tracking numbers (1Z9999W99845399981)
- Often have embedded spaces
- Older IR systems may not index numbers

  But often very useful:
  - looking up error codes/stack traces on the web
  - Date of an email Will often index “meta-data” separately, Creation date, format, etc.

Tokenization

- Tokenization: language issues
- French
  - **L'ensemble** one token or two?
  - L ? L' ? Le ?
- German noun compounds are not segmented
  - **Lebensversicherungsgesellschaftsangestellter**
  - 'life insurance company employee'
- Chinese and Japanese have no spaces between words

  莎拉波娃现在居住在美国东南部的佛罗里达。

  - Not always guaranteed a unique tokenization
- Arabic (or Hebrew) is written right to left, but with certain items like numbers written left to right
- Use rule-based or machine learning-based *compound-splitters* or *word segmentation* tools to tokenize long compound words or languages where explicit separators are not used to indicate word boundaries.

Stop words

- Common words that appear to be of little value in helping select documents matching a user's need.
- With a stop list, exclude from the dictionary entirely the most common.
- They have little semantic content
    - *the, a, and, to, be*
- To sort the terms by *collection frequency* and then to take the most frequent.

Issues in removing stop words

- Some special query types are disproportionately affected.
  - Phrase queries:
    - “King of Denmark”
    - “President of the United States”, President AND “United States”
  - Various song titles, etc.:
    - “Let it be”, “To be or not to be”
  - “Relational” queries:
    - “flights to London”: if *to* removed, it implies both “flights to London” or “flights from London”
- Standard use of quite large stop lists (200–300 terms) to very small stop lists (7–12 terms)

Token normalization

- *Token normalization* is the process of canonicalizing tokens so that matches occur despite superficial differences in the character sequences of the tokens
  - match U.S.A. and USA
- A term is a (normalized) word type, which is an entry in the IR system dictionary
- To implicitly create **equivalence classes**, which are normally named after one member of the set.
  - deleting periods to form a term
    U.S.A., USA
  - deleting hyphens to form a term
    anti-discriminatory, antidiscriminatory

Token normalization

- Alternatives to creating equivalence classes are
  - to maintain relations between unnormalized tokens.
  - to do asymmetric expansion.
  - Example: Microsoft Windows, Rear Window, glass window
    - Enter: **window** Search: **window, windows**
    - Enter: **windows** Search:**Windows,windows, window**
    - Enter: **Windows** Search: **Windows**

Maintain relations between unnormalized tokens

1. Index unnormalized tokens.
2. Maintain a query expansion list of multiple vocabulary entries to consider for a certain query term.
3. A query term is then effectively a disjunction of several postings lists.

Asymmetric expansion

Perform the expansion during index construction e.g. When the document contains *automobile*, we index it under *car* as well and vice versa.

Token normalization

- Accents and Diacritics: Naïve, pe*ñ*a (a cliff), pena (sorrow).
- Case folding – True Casing
- Reduce all letters to lower case
- The simplest heuristic is to convert to lowercase words
    - at the beginning of a sentence
    - all words that are all uppercase or in which most or all words are capitalized
    - exception: upper case in mid-sentence

Text normalization

- Handling synonyms and homonyms
  - e.g., by hand-constructed equivalence classes
    - 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 expand a query
    - When the query contains **automobile**, look for **car** as well
- Spelling mistakes
  - One approach is Soundex, which forms equivalence classes of words based on phonetic heuristics

Stemming and Lemmatization

- To reduce inflectional forms and sometimes derivationally related forms of a word to a common base form.
- Example:
  - *am, is, are* → *be*
  - *car, cars, car's, cars'* → *car*
  - *the boy's cars are different colors* → *the boy car be different color*

Stemming and Lemmatization

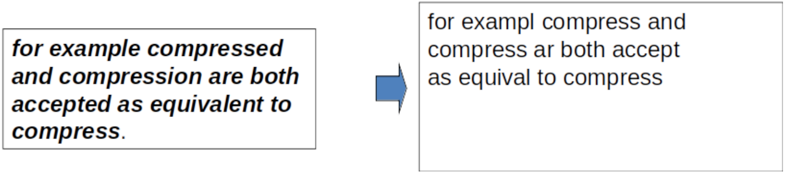
- **Stemming** refers to a crude heuristic process that chops off the ends of words and removes the derivational affixes.

  It commonly collapses derivationally related words
- **Lemmatization** refers to doing things properly with the use of a vocabulary and morphological analysis of words, normally aiming to remove inflectional endings only and to return the base or dictionary form of a word, which is known as the lemma.

  It only collapses the different inflectional forms into the corresponding root forms.

Stemming

- Reduce terms to their common basic form before indexing.
- "Stemming" suggests crude affix chopping
  - 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 compress and compress ar both accept as equival to compress

Porter's Stemmer

- The most common algorithm for stemming English.
  - Results suggest it's at least as good as other stemming options
- Algorithm
  1. 5 phases of reductions
  2. phases applied sequentially
  3. each phase has various conventions to select rules
  4. **sample convention:** Of the rules in a compound command, select the one that applies to the longest suffix.
- **Phase 1**

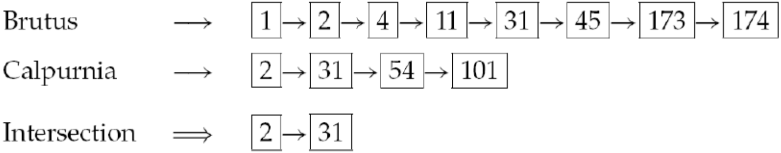
SSES → SS	caresses → caress
IES → I	ponies → poni
SS → SS	caress → caress
S →	cats → cat

- **Phase 2**
  - Loosely checks the number of syllables to find whether a syllable is suffix or part of the stem of the word.
  - *replacement*→*replac*, and NOT *cement* → c

Lemmatizer

- Tool from Natural language Processing
- Does full morphological analysis to accurately identify the lemma for each word.
- Full morphological analysis
  - Is usually more time consuming and elaborate process.
  - produces at most very modest benefits for retrieval.

Faster postings list access



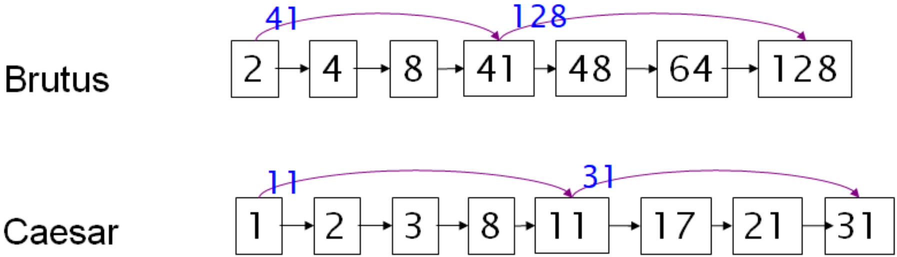
Brutus ⟶ 1 → 2 → 4 → 11 → 31 → 45 → 173 → 174

Calpurnia ⟶ 2 → 31 → 54 → 101

Intersection ⟹ 2 → 31

- If lengths of *postings lists* are *m* and *n* then, intersection operation takes $O(m+n)$ time.
- The speed of intersection may be increased by using **skip pointers**
- **Skip pointers** are shorcuts to bypass parts of *posting lists* that will not appear in the search result

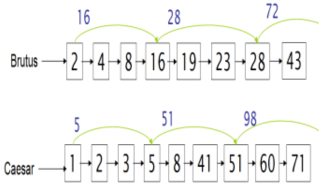
Skip pointers

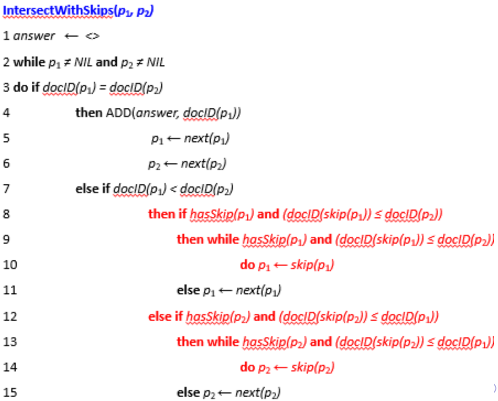


- **Points to consider**

  Where to place the skip pointers?
  How to do efficient merging using skip pointers?

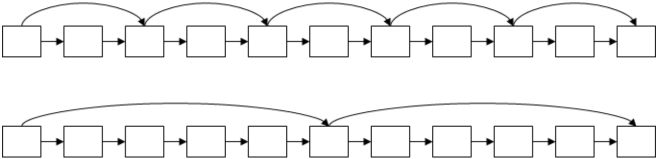
Skip pointers





```
IntersectWithSkips($p_1$, $p_2$)
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 if $hasSkip(p_1)$ and $(docID(skip(p_1)) \leq docID(p_2))$
                   then while $hasSkip(p_1)$ and $(docID(skip(p_1)) \leq docID(p_2))$
                             do $p_1 \leftarrow skip(p_1)$
                   else $p_1 \leftarrow next(p_1)$
           else if $hasSkip(p_2)$ and $(docID(skip(p_2)) \leq docID(p_1))$
                   then while $hasSkip(p_2)$ and $(docID(skip(p_2)) \leq docID(p_1))$
                             do $p_2 \leftarrow skip(p_2)$
                   else $p_2 \leftarrow next(p_2)$
```

Where to place the skip pointers?



- **More skips → shorter skip spans**
  1. more likely to skip
  2. increased number of skip comparison operations.
  3. more successful skips
- **Less skips → longer skip spans**
  1. fewer pointer comparison
  2. fewer successful skips
- **Simple heuristic:** for postings of length P, use $\sqrt{P}$ evenly-spaced skip pointers.

Phrase queries

- Consider the query - "**Stanford University** - as a phrase
- Following documents are *false positives*
  1. *"I went to university at Stanford"*
  2. *"The inventor Stanford Ovshinsky never went to university"*
- Postings lists comprising of **documents containing individual terms** not sufficient to handle such queries.
- Approaches for phrase queries
  1. **Biword Indexes**
  2. **Positional Indexes**

Biword Indexes

- Index every consecutive pair of terms in the text as a phrase
- *Query:* **"Friends, Romans, Countrymen"**
- Pairs of consecutive words indexed as dictionary terms
  - *friends romans*
  - *romans countrymen*
- For longer queries consecutive word pairs are *AND*ed
  1. Query: **"Friends, Romans, Countrymen"**
     (friends roman) AND (roman countrymen)
  2. stanford university palo alto
     (stanford university) AND (university palo) AND (palo alto)
- **Disadvantage:** False positives: The *biwords* may not necessarily appear together in the retrieved document.

Extended biwords

- **Nouns and noun groups (N)** are usually *more significant* in queries as compared to words with other parts-of-speeches (X).
- For any string of terms of the form NX^*N, the word pair corresponding to NN
  - forms an **extended word pair**
  - indexed in the dictionary

cost	overruns	on	a	power	plant
N	**N**	**X**	**X**	**N**	**N**

- Extended bi-words
  1. cost overruns
  2. overruns power
  3. power plant
- **Query:** *(cost overruns)* AND *(overruns power)* AND *(power plant)*

Positional indexes

- Store in the posting the **positions where the term appear** in the document.

<**term**, # docs containing **term**;

doc1: freq. of the term; pos1, pos2, · · · ;

doc2: freq. of the term; pos1, pos2, · · · ;

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); · · ·)

Proximity intersection

- **Query:** *to be or not to be*
- Start from the postings lists of the terms in *increasing order* of document frequency.
- Consider **to** and **be**
  1. Find the documents containing both terms
  2. Look for positions in the lists where **be** occurs with **one index position greater** than an occurrence of *to*
  3. Look for occurrence of both words with token positions 4 higher than first occurrence

to: $< \cdots$;4:$< \cdots$,429,433,$>$;$\cdots >$

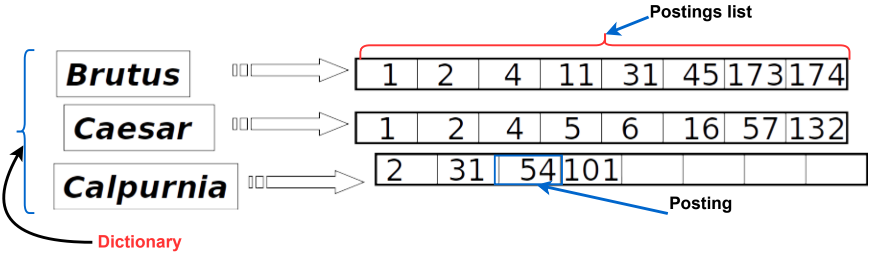
be: $< \cdots$;4:$< \cdots$,430,434,$>$;$\cdots >$

Dictionaries

- Dictionary data structures
- Tolerant retrieval
    1. Wild-card queries
    2. Spelling correction
    3. Phonetic correction
- Develop techniques that are robust to typographical errors in the query, as well as alternative spellings.

Search structures for dictionaries

- The **dictionary data structure** stores the term vocabulary, document frequency, pointers to each postings list.
- Explore the data structures for the dictionary.



A simple dictionary

- An array of structures

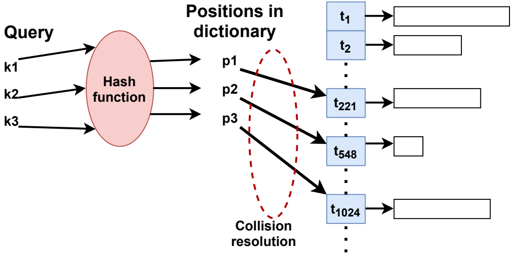
term	document frequency	pointer to postings list
a	656,265	→
aardvark	65	→
...	...	...
zulu	221	→
char[20]	int	Postings
20 bytes	4/8 bytes	4/8 bytes

- **Storage and retrieval is not efficient**
- Points to be considered:
  1. # of terms in dictionary
  2. Keys remain static or dynamic
  3. The relative frequencies with which various keys will be accessed
- Two choices
  1. **Hashtables**
  2. **Trees**

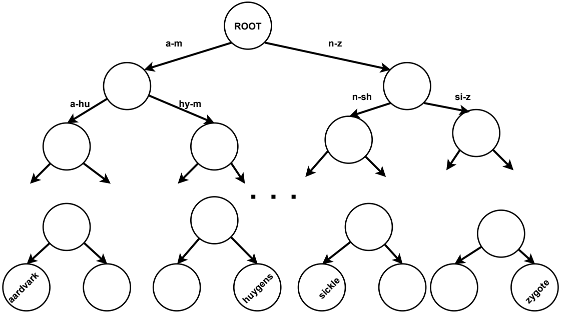
Hashing

- Query terms (keys) mapped to integers from a big enough space to avoid collision.
- Collision resolution done by *auxiliary structures*
- $O(1)$ search complexity

- **Cons**
  1. Minor variants may be mapped to distant integers.(color/colour)
  2. No prefix search (free/free**ly**/free**dom**)
  3. Expanding vocab may necessitate *redesigning the hash function*.

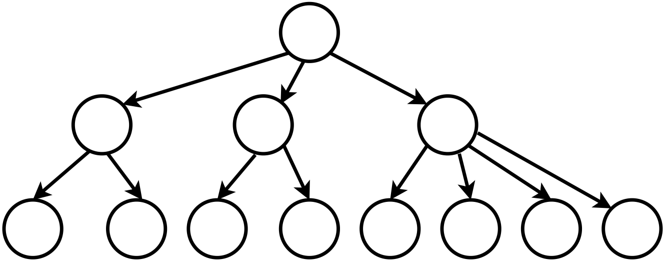


Binary trees



- Efficient search time is $O(M)$ if tree is balanced
- Allows **prefix search**
- If balanced at each node, the difference in depth of left and right subtrees differ by at most 1.
- Insertion and deletion unbalance a tree
- Costly **rebalancing** step required to maintain balance

B-tree



- To mitigate rebalancing, B-trees may be used
  - Each *internal node* of a **B-trees** has variable number of children in a fixed range.
  - Each branch under an internal node represents a test for a range of character sequences.