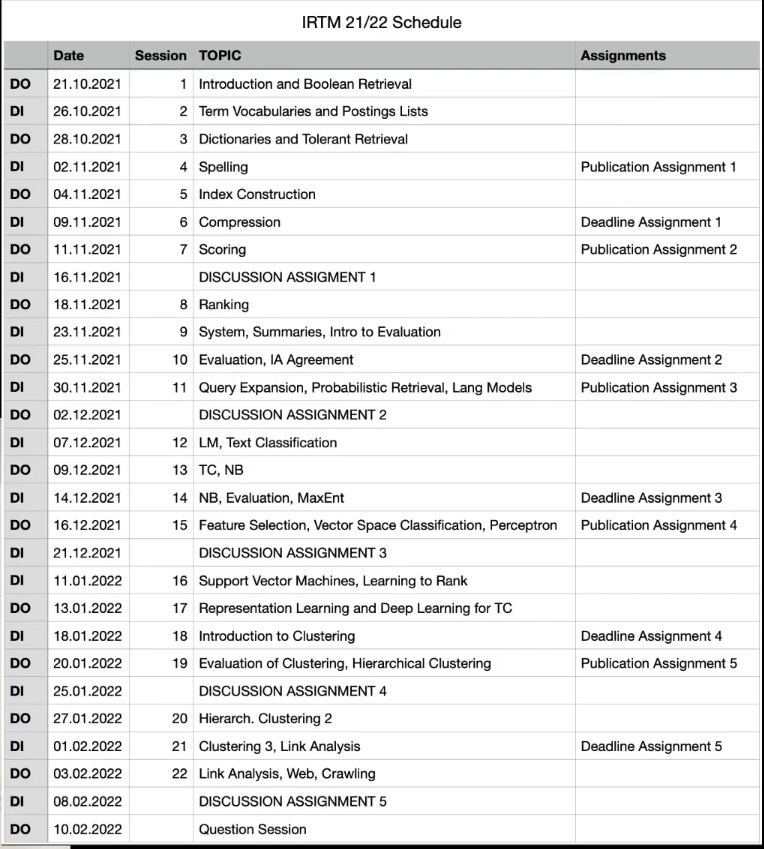
# Information Retrieval & Text Mining



* Information Retrieval
* Text Mining:
  + Extract information from unstructured text
* Book:
  + http://informationretrieval.org

# Formalities

* Lectures on Tuesdays and Thursdays
* 55.01 🡪 videos uploaded in the evening
* 5 home work assignments!
* Register in Campus:
  + Assignments (Hausübungen)
    - 80 points in assignments to write exam.
    - 100 points in pen&paper exercises.
    - 50 points in practical exercises 🡪 points granted for submission of correct and well documented code/result.
    - Do it alone.
  + Written exam (not used because Vertiefungslinie)
  + Vertiefungslinie

# Lecture 1 - Boolean Retrieval

What is information retrieval?

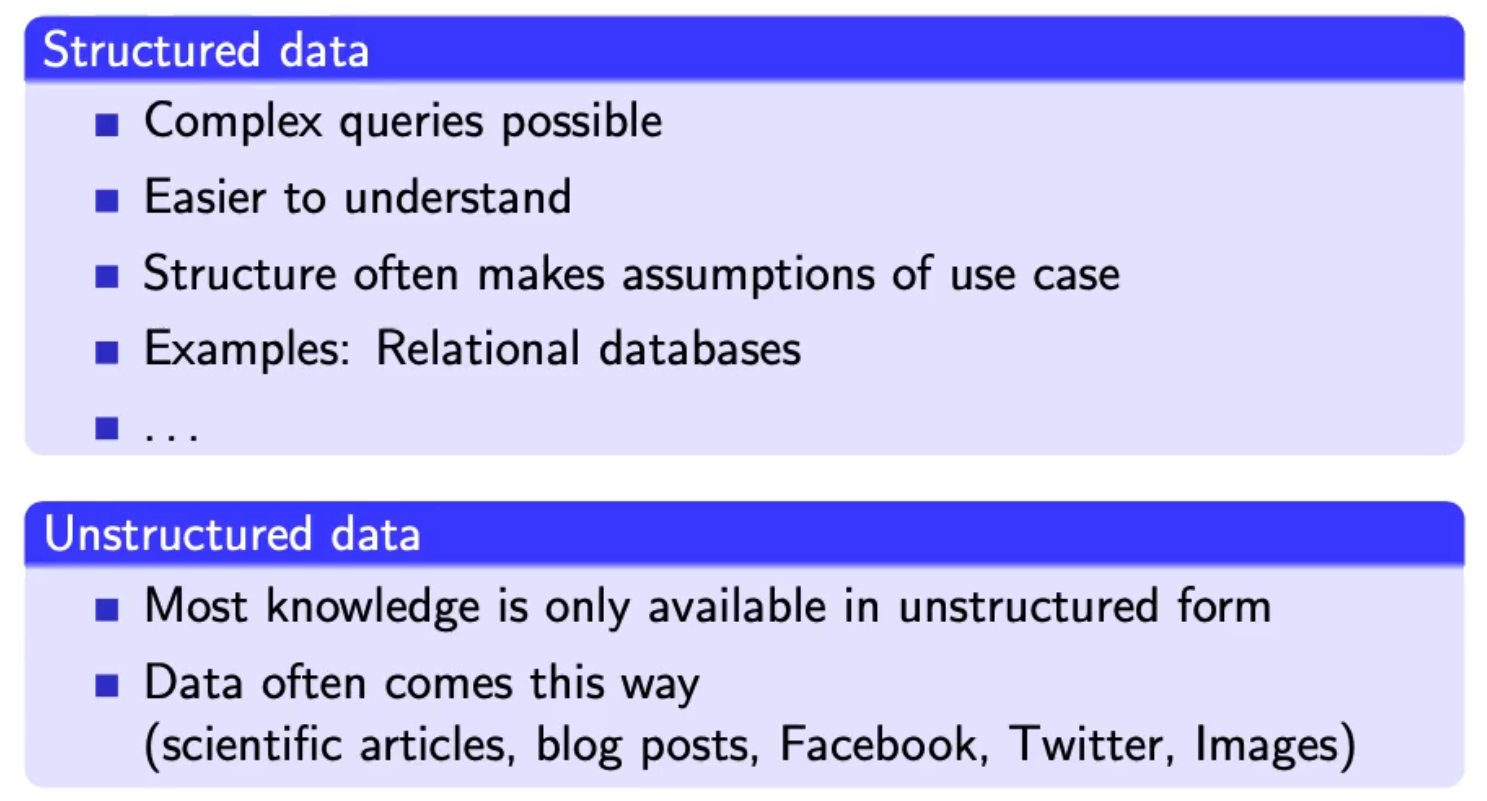
* IR is finding material (documents) of an unstructured nature (text) that satisfies an information need from within large collections, usually stored on computers.
* Satisfies an information need.

What is Text Mining?

* Text mining is the derivation of information (structured form) from unstructured text. In contrast, data mining is typically applied on structured data. Typically, methods from information retrieval are used.
* Conversion process from text to data.

Why there is more unstructured data in contrast to structured databases?

* No need to structure data 🡪 blog posts
* Text is a natural form on communication
* Text is more accessible
* Structuring data is expensive



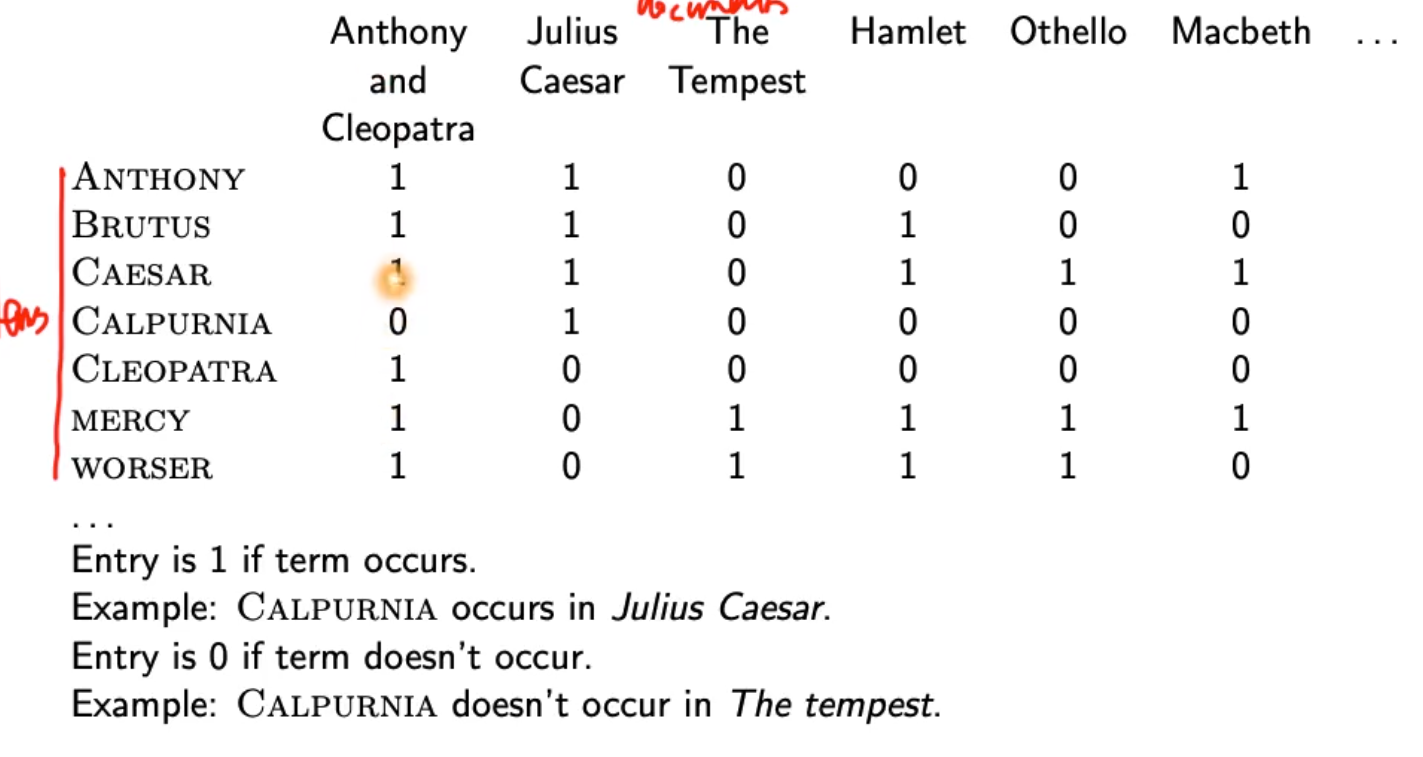
Boolean retrieval:

* Boolean model: simplest model to base an information retrieval systemon
* Queries and boolean expression e.g. Caesar and Brutus
* Search engine returns all documents that satisfy the Boolean expression
* Does Google use the Boolean model?
  + Query expansion with synonyms
  + Cases where you get hits that do not contain one of the words:
    - Anchor texts
    - Page contains variant of word (synonym, spelling correction)
    - Long queries (n large)
    - Boolean expression generates very few hits
* Simple boolean retrieval returns matching documents **in no particular order.**
* Most Boolean engines rank the result set – they rank good hits (**according to some estimator of relevance**) higher than worse hits.

## Inverted Index

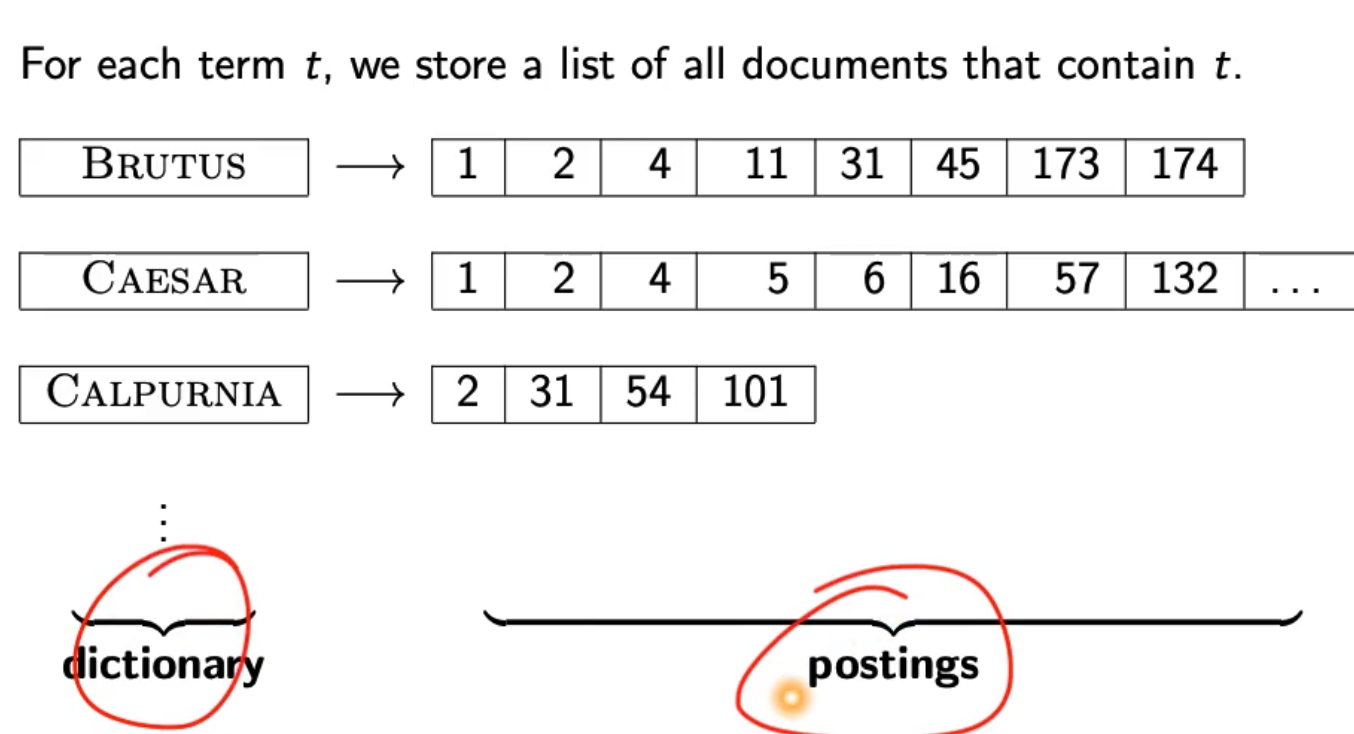
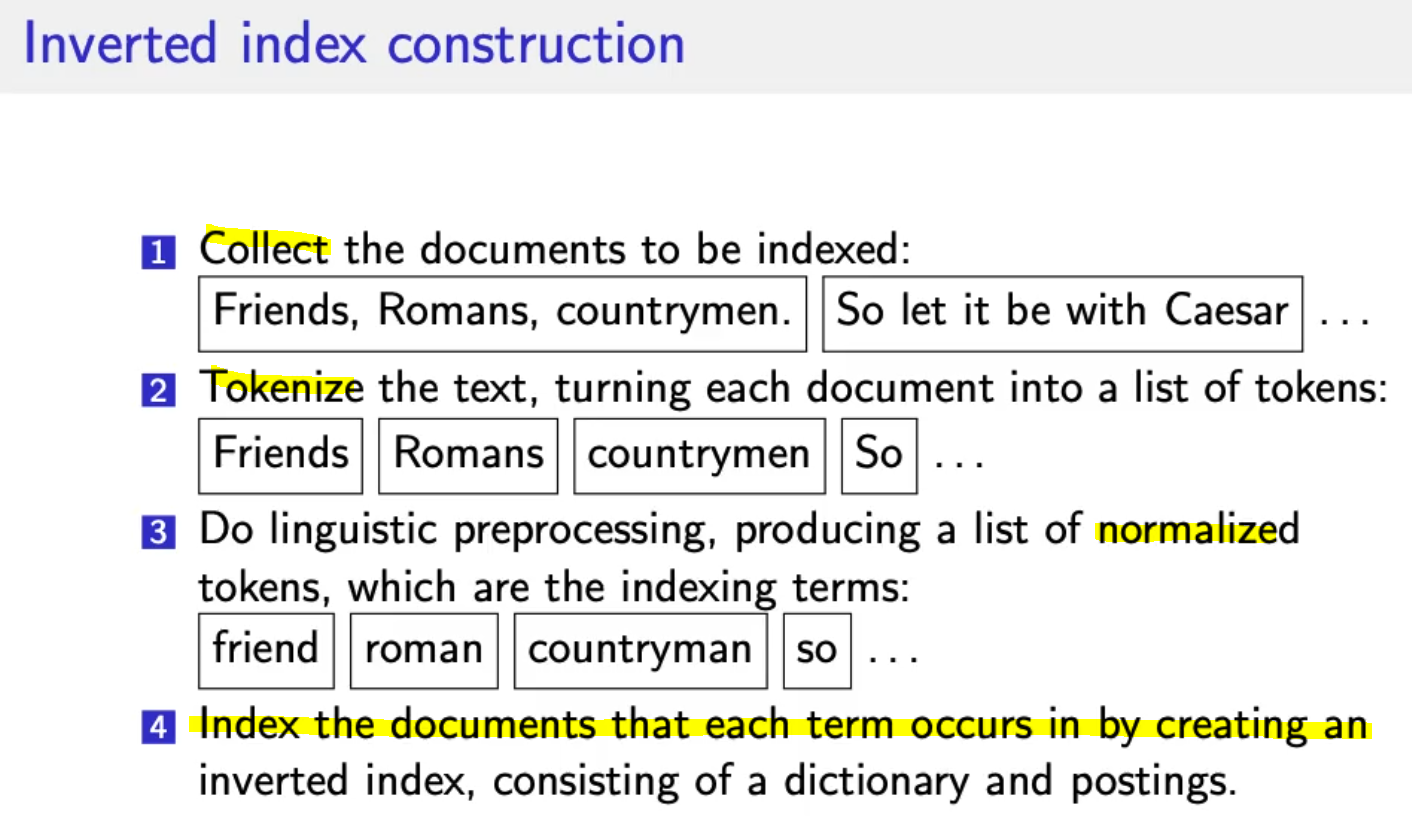
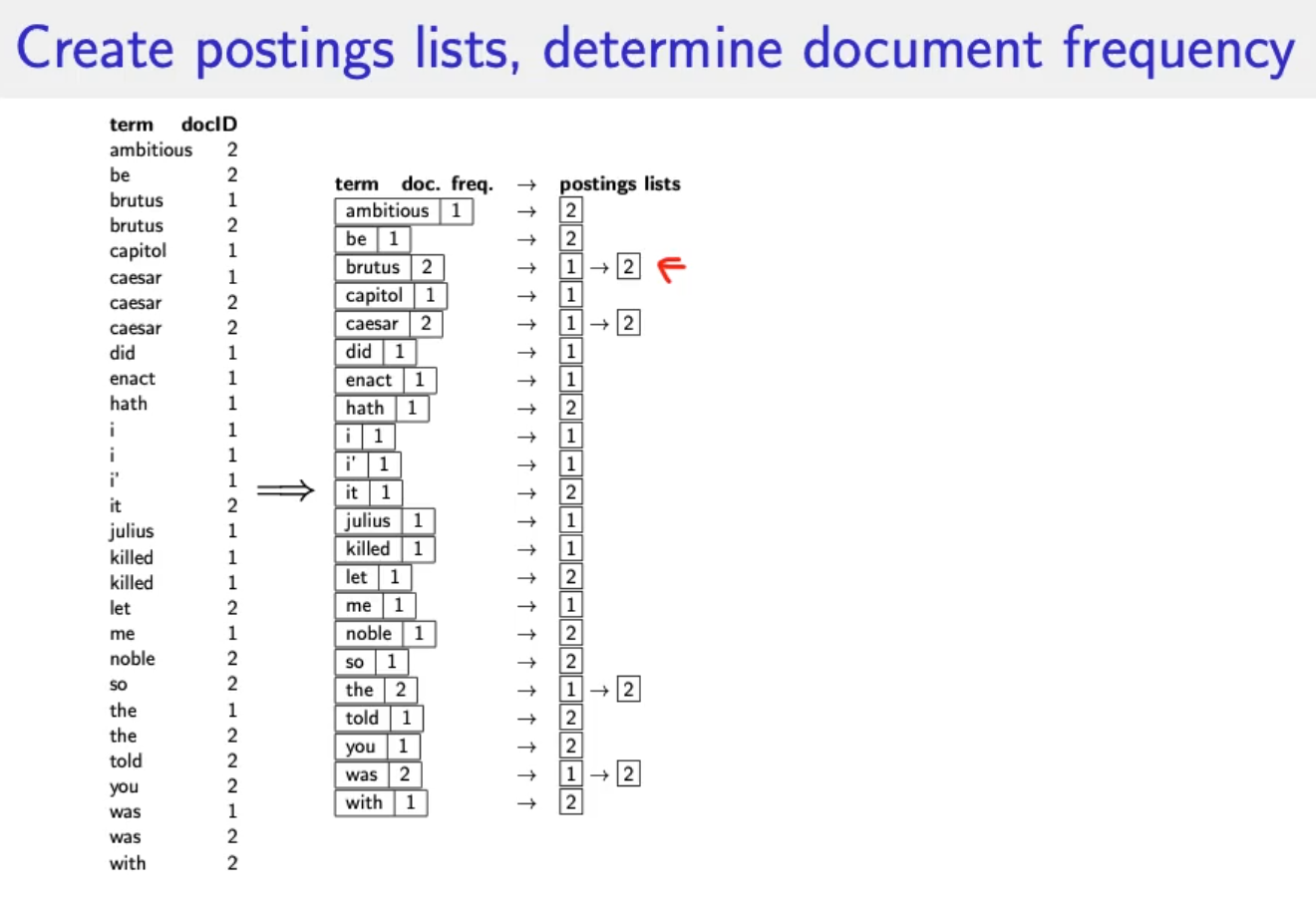
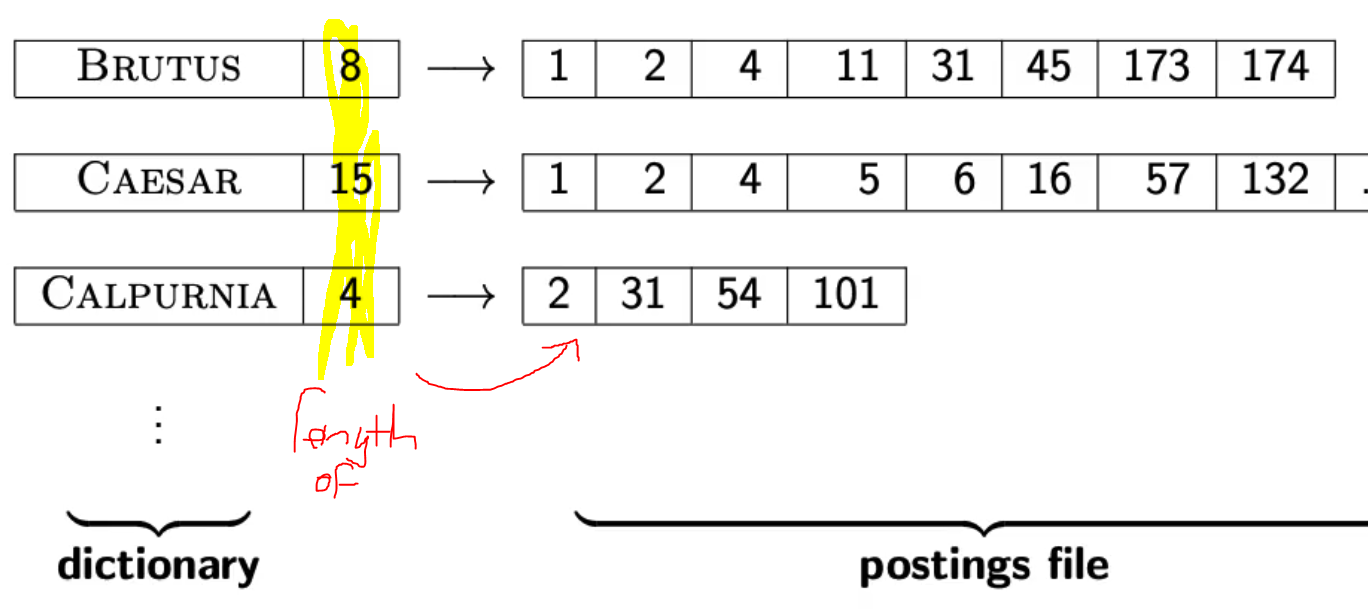
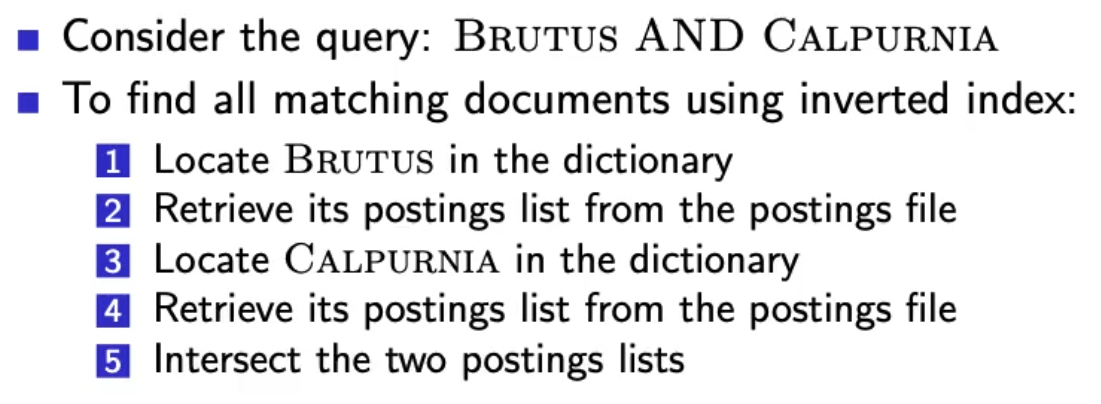
* Which plays of Shakespare contain the words Brutus & Caesar but NOT Calpurnia?
  + grep “Brutus” | grep “Caesar” | grep -v “Calpurnia”
    - inefficient, scan all documents 3 times
    - grep doesn’t know what a “word” is, Caesar-Salad would also be a hit.
    - Grep is line oriented, IR is document-oriented
    - Other operations 🡪 find close words “romans” / “countrymen”

## Term-document incidence matrix

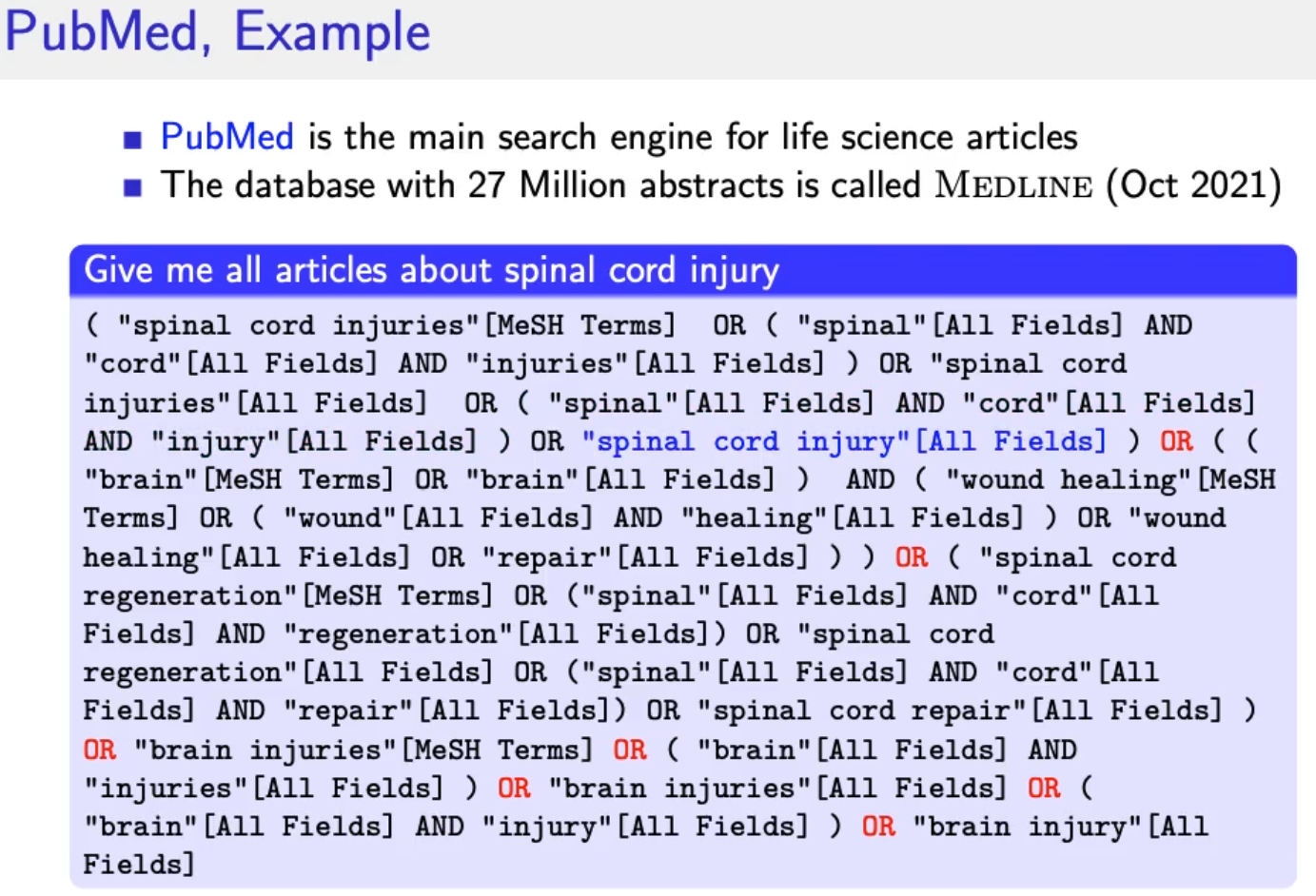


* Can’t build incidence matrix for big collections (1 million docs, 1000 words per doc 🡪 half trillion 0/1)
* Extremely sparse matrix 🡪 only record the ones

## Inverted index

* Point from terms to documents:
* 
* 
* 
* 
* 
* Runtime of intersecting the two lists?
  + Linear in the sum length of the postings lists
  + This only works if posting lists are sorted



* Boolean retrieval model can answer any query that is a boolean expression
  + AND, OR, NOT
  + Views each document as a set of terms
  + Precise: Document machtes or not, nothing in between
  + 🡪 Primal retrieval tool for at least 3 decades
  + Many professional searches (lawyers, patent offices, …) prefer boolean queries
    - You know exactly what you are getting
    - No Google “magic”
  + Westlaw: big law search engine
    - /3 within 3 words
    - /s within a sentence
    - /p within a paragraph
    - Long precise queries, incrementally developed not like a web search
  +  🡪 needs query optimization

## Query Optimization

* Consider a query that is and of n terms, n>2
* For each of the terms get postins list, then AND them together
* Example query:
  + BRUTUS and CALPURNIA and CAESAR
  + 🡪 check the shortest posting list and start with these 🡪 fewer comparisons in the next step
* Example query:
  + (MADDING or CROWD) and (IGNOBLE or STRIFE)
  + Get frequencies of all terms
  + Estimate the size of each or by the sum of its frequencies (conservative)
  + Start with shortest

# Lecture 2 – Voc

## Documents

* Spoken language
* Handwritten texts
* Scanned documents
* Paper documents
* Signed language (Gebärdensprache)
* Format / Language
  + Format pdf, excel, html
  + Language
  + Character set
  + General approach:
    - Converter for each format to a generic format
  + A single index/document contains terms of several languages
    - French email with spanish pdf attachment
  + Document unit?
    - 1 file
    - 1 email
    - Mail + all attachments
    - Group of files (LaTex)
    - What about XML?
* Issues
  + Mathematical equations in documents
  + OCR/Scan 🡪 Which column belong together?
  + Hyphenization 🡪 detect and put words together
  + HTML pages (Advertisements, Images)
  + Issues with proprietary formats

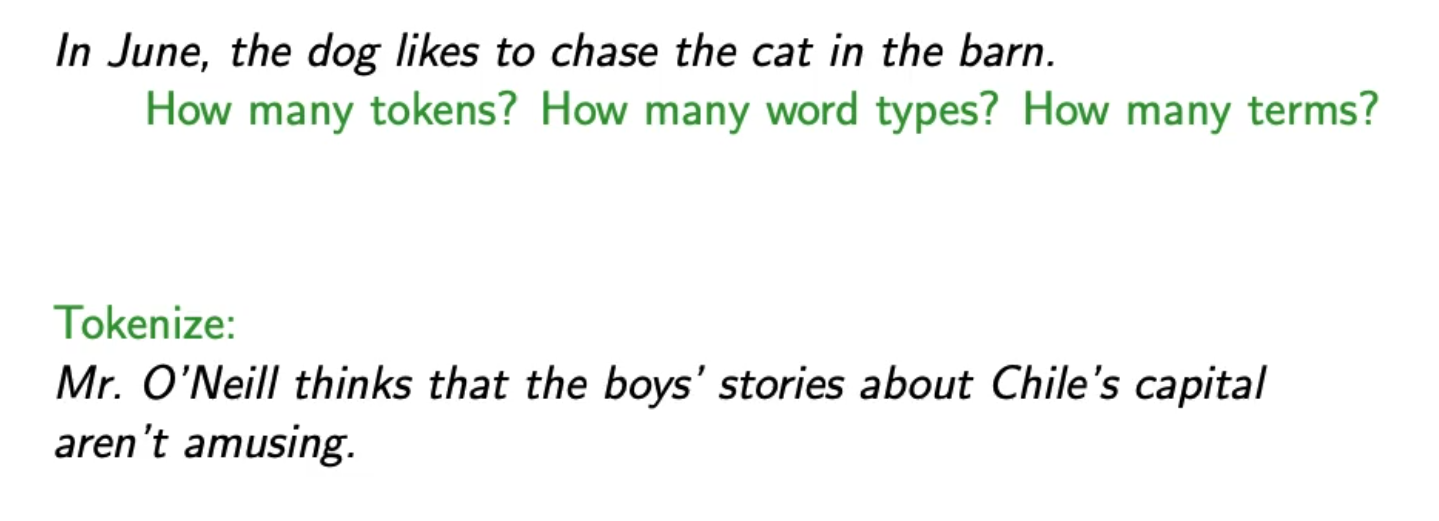
## Terms

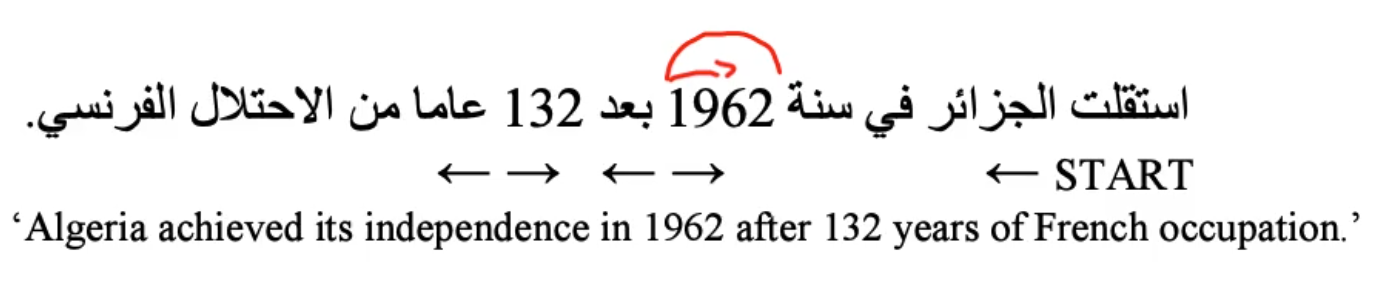
* Token: Character sequence in a document
  + Closely related to one Word
  + Punctuation is also a token != word
* Type: equivalence class of tokens
  + Related to Term: (normalized) type 🡪 thing we put in dictionary
* Term: as it occurs in IR system’s dictionary
* Example:
  + “I like the coffee, cofees, and the shop.
  + How many tokens? Types? Terms?
  + 11 tokens
  + 9 types (commas and the, the)
  + 8 terms 🡪 plural coffee and coffees

## Normalization

* + Normalize words in indexed text as well as query terms into the same form
  + U.S.A. = USA, coffees = coffee
  + Two approaches:
    - **Implicitly define equivalence classes of terms (rules)**
    - **Asymmetric expansion**
      * Window -> window, windows
      * More powerful but less efficient
  + Normalization process and language detection interact
    - PETER WILL NICHT MIT! 🡪 MIT = mit
    - He got his PhD from MIT. 🡪 MIT != mit

Übung:



* First example:
  + 14 tokens
  + 12 types (In != in), (the=the=the)
  + 11 terms
* Second example:
  + Mr. | O’Neill | thinks | that | the | boys’ | stories | about | Chile’s | capital | aren’t | amusing | . |
* Tokenization problems:
  + Hewlett-Packard = one word
  + State-of-the-art = adjective
  + co-education
  + the hold-him-back-and-drag-him-away maneuver = ineffiecnet to get all words
  + data base
  + San Francisco = one token
  + Los Angeles-based company 🡪 we need a dictionary of names
  + Cheap San Francisco-Los Angeles fares
  + York University vs. New York University (what if search for york university and we find new york?)
  + Numbers
    - 3/20/91 🡪 know what it is
    - 20/3/91
    - Mar 20, 1991
    - Normalize dates to the same format
    - B-52 🡪 Bomber
    - 100.2.86.144 🡪 ip address?
  + Chinese:
    - No whitespace
    - Compounds in German 🡪 Donaudampfschifffahrtsgesellschaft
  + Arabic script: bidirectionality
    - 
  + Accents:
    - Résumé vs resume
  + Umlaute
    - Universität vs Universitaet
  + How are users likely to write their queries for these words?
  + Even in languages that standardly have accents, users often do not type them e.g. Polish.

### Normalization of English

* Case folding 🡪 all letters to lowercase
* Even though case ca be meaningful
  + MIT vs. mit
  + Fed vs fed vs FeD vs FED
* Best to lowercase everything
* Counter example:
  + Human gene CES4A
  + Rat gene name Ces4a (by convention only first character)

### Stopwords

* Extremely common words which would appear to be of little value in helping selecting documents matching user need
* A, an, and, the, was, to ,that, on, from, for
* Stop word elimination used to be standard in IR systems
* Stop words for phrase queries 🡪 King of Denmark
* Most web search engines index stop words

### Soundex

* Equivalence classes based on phonetics
* Muller = Mueller

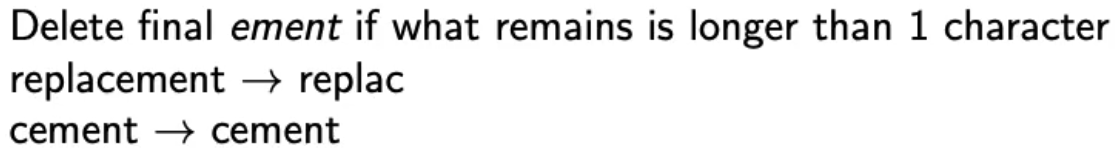
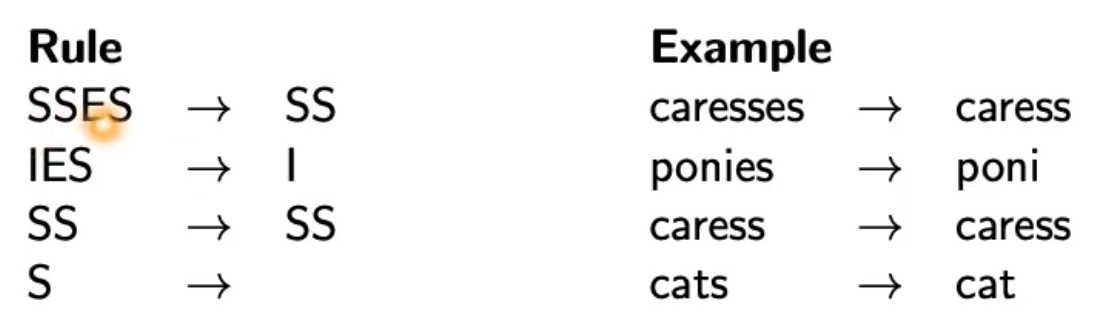
### Thesauri / Ontologies

* Semantic equivalence 🡪 car = automobile

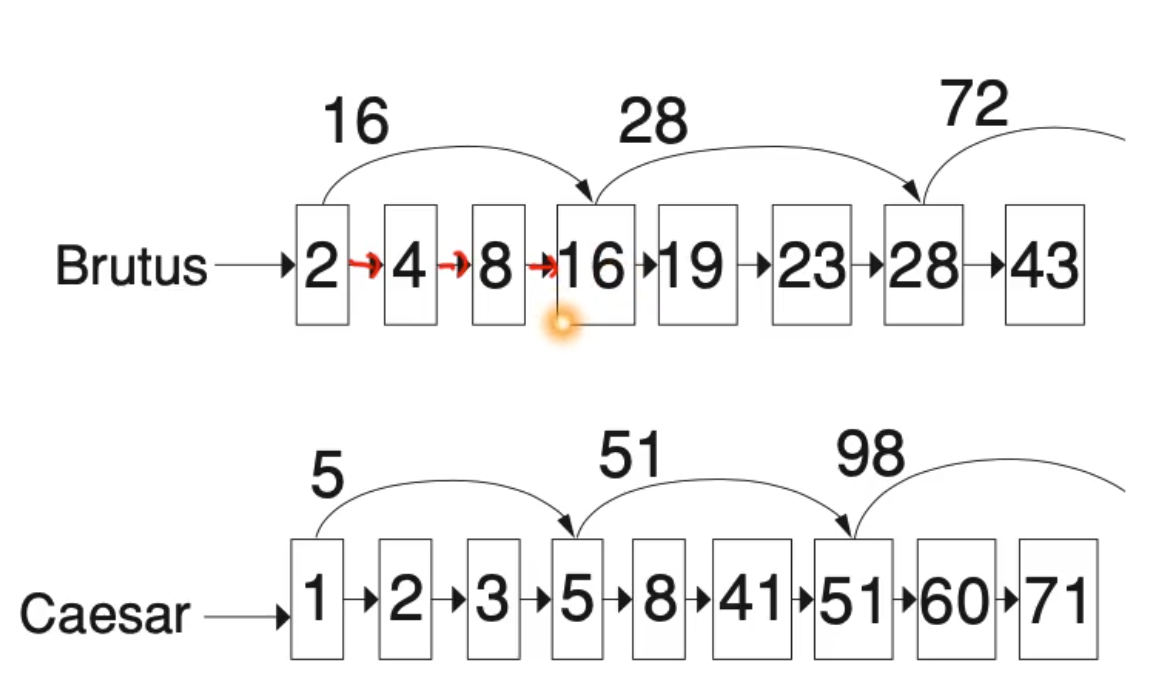
### Lemmatization

* Reduce inflectional/variant forms to base form
* Am, are, is 🡪 be
* Car, car, car’s 🡪 car
* The boy’s cars are different colors 🡪 the boy car be different color
* Lemmatization implies doing proper reduction to dictionary headword
* Inflectional morphology:
  + Cutting 🡪 cut
* Derivational morphology:
  + Destruction 🡪 destroy

### Stemming

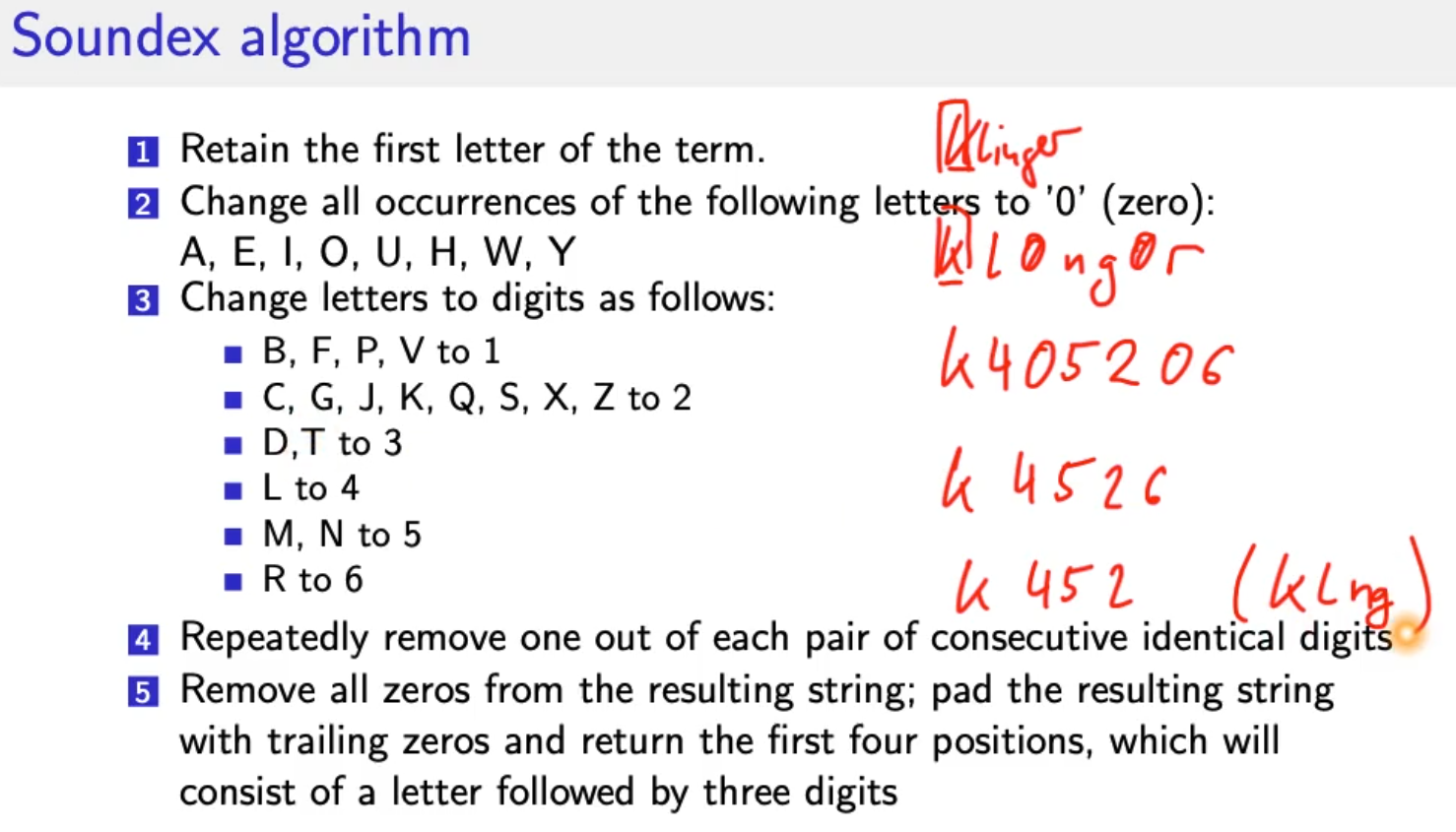
* Heuristic process that chops off the ends of words
* Achieve “principled” lemmatization
* Language dependent
* Inflectional AND derivational
* Example for derivational (other meaning)
  + Automate, automatic, automation 🡪 automat
* Porter algorithm most common algorithm for stemming in English
  + Conventions and 5 phases of reductions
  + Phases are applied sequentially
  + Each phase consists of a set of commands
  + Sample command:
    - 
  + Always apply the longest rule 🡪 applies to the longest suffix
  + Subset of rules:
    - 
* Different stemmers lead to different results
* Sometimes stemming increases effectiveness, sometimes decreases!

## Skip pointers

* Skip pointers enable intersection algorithm to skip postings that will not figure in the search results
* Intersecting becomes more efficient
* 
* Tradeoff
  + Number of skiped items
  + Frequency how often can we skip?
    - More skips: skip pointer only skips few items but we can frequently use it.
    - Fewer skips: skip pointer skips many items but we can’t use it very often.
* Simple heuristic:
  + For postings list of length P use sqrt(P) evenly-space skip pointers
  + This ignores the distribution of query terms.
* Easy if index is static, harder when index is dynamic!

# Lecture 3 – Phrase Queries, Dictionaries

## Soundex

* Soundex is basis for finding phonetic (as opposed to orhographic) alternatives
* Example: chebyshev / tchebyscheeff
* Algorithm:
  + Turn every token to be indexed into a 4-character reduced form
  + Do the same with query terms
  + Build and search an index based on these reduced forms
  + 
* For general information retrieval not very useful
* Ok for high recall tasks (Interpol)
* HASENBALG
* H0S0NB0LG
* H02051042
* H251 🡪 HSNB

## Phrase queries

* Stanford university as phrase
* “The inventor Stanford Ovshinksky never went to university” 🡪 should not be a match
* Easily understood concept for end users (only 10% are explicit phase queries)
* Consequence for inverted index 🡪 it no longer suffices to store docIds in postings lists
* Three remedies:

### Biword index (phrase index)

* + - * Index every consecutive pair of terms in the text as phrase
      * Each biword is now vocabulary term
      * Two-word phrases can easily be answered
      * 🡪 increase tri-word, quad-word index etc. but inefficient
      * Why are biword indexes rarely used?
        + Index blowup
        + False positives

### Positional index

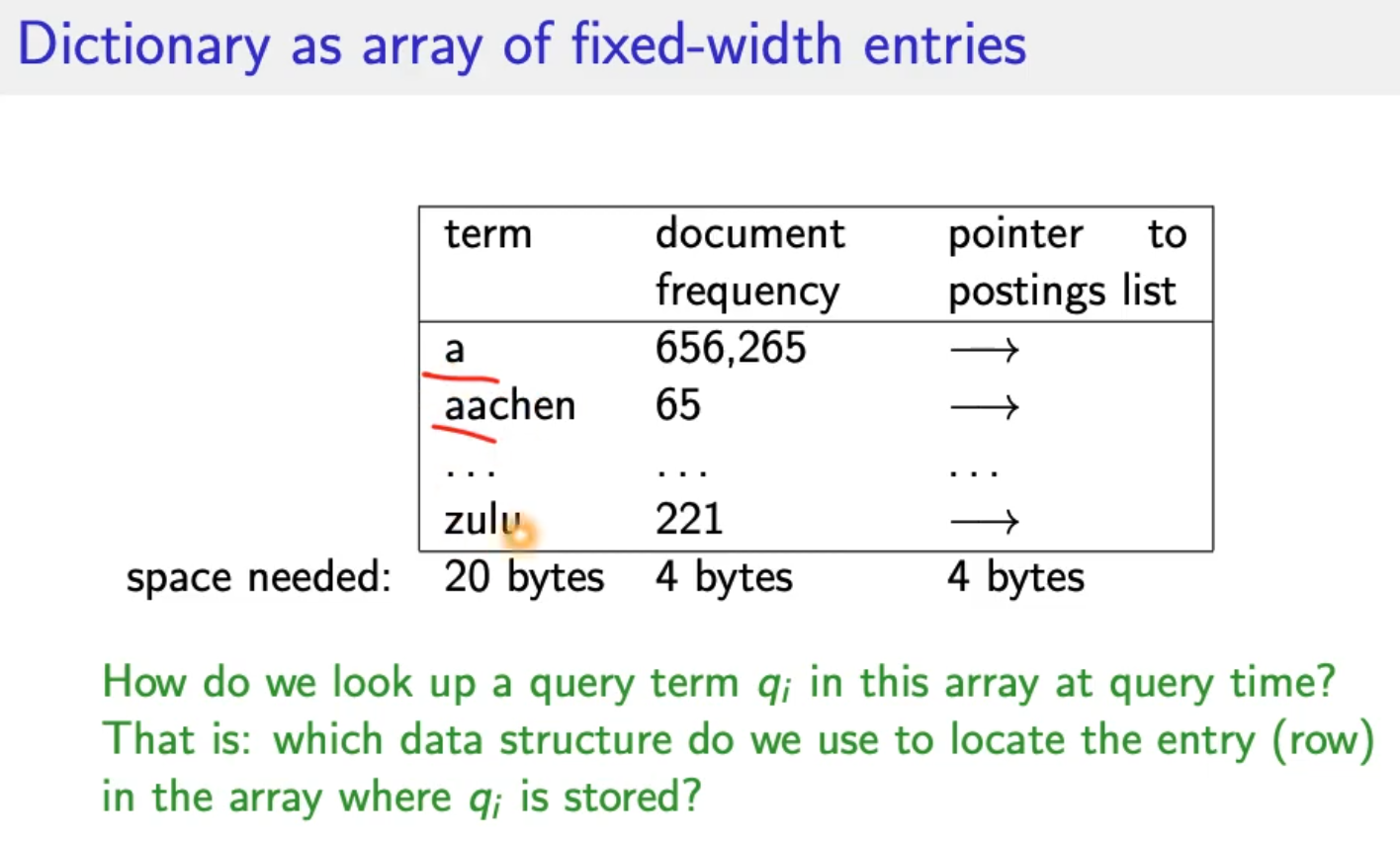
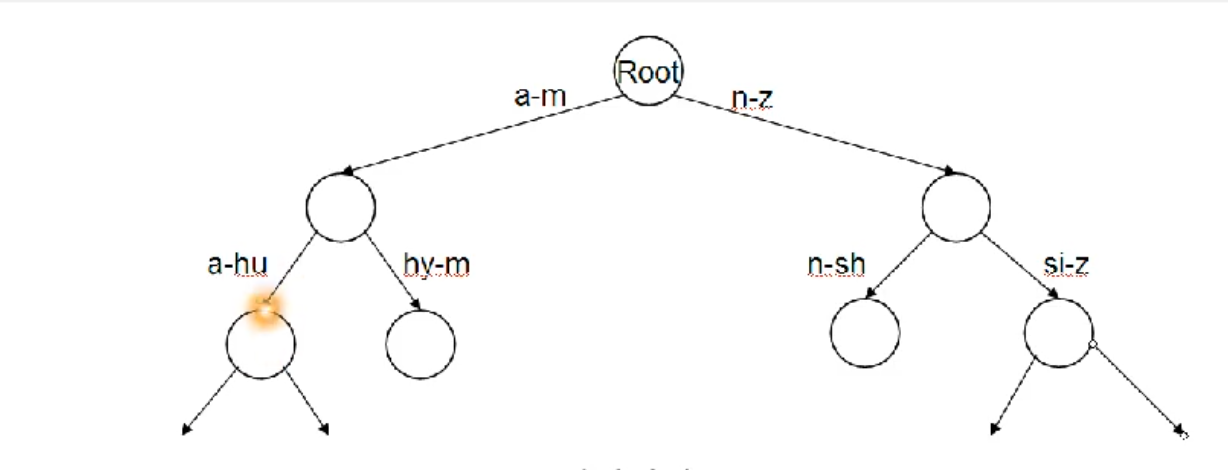
* + - * More efficient
      * Postings list in a positional index:
        + Each posting is a doc Id and a list of positions (where does the term occur in the document?)
      * Query: compare list of positions of postings, if order matches it’s a hit
      * Also use it for proximity search (comparison value = 4)
        + Employment /4 place
      * Google:
        + Test \* Data 🡪 low distance
        + Test \*\*\*\* Data 🡪 high distance
      * Show actual matching positions to user!

### Combination of biword/positional

* + - * Many biwords are extremely frequent:
        + Micheal Jackson, Britney Spears
      * For these biwords increased speed compared to positional postings intersection is substantial!
      * Frequent biword included in index, everythin else positional index!

**Positional queries are much more expensive than regular Boolean queries.**

## Dictionaries

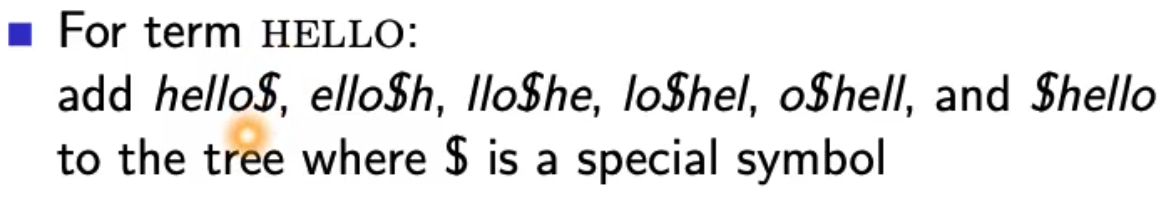
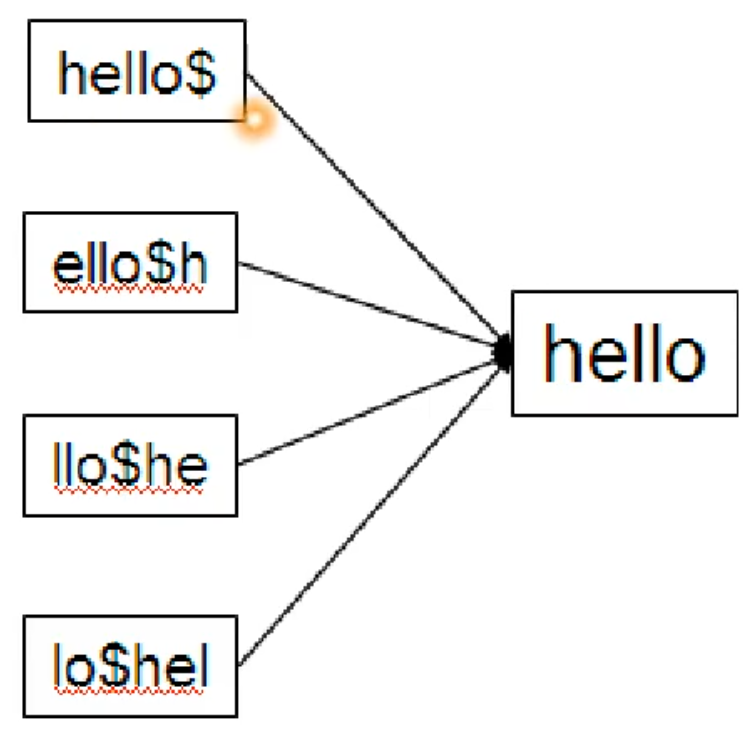
* How to store data structure
* Dictionary: data structure to store term vocabulary
* Term vocabulary: the data
* For each term we need to store a couple of items:
  + Term
  + Document frequency
  + Pointer to postings list
  + Assume we can store this info in a fixed-length entry!
  + We store these entries as array.
  + 
* Data structures for looking up term
  + Criteria:
    - Is there a fixed number of terms or will it keep growing?
    - What are the relative frequencies with which various keys will be accessed?
    - How many terms are we likely to have?
  + Hashes
    - Each vocabulary term is hashed into an integer
    - This number is the row number of the array
    - At query time: hash(query term) = row 🡪 super quick, runtime O(n), faster than tree
    - Disadvantage:
      * no way to find minor variants (resume vs résumé) 🡪 Hash function doesn’t know about language!
      * Lookup time depends on hash function
      * No prefix search (all terms starting with automat)
  + Trees
    - Solves prefix problem 🡪 find all terms starting with automat
    - Simplest tree: binary tree
    - Slower than hashes O(log M) (holds for balanced trees) 🡪 M = size of vocabulary
    - Rebalancing binary trees is expensive
    - B-trees address the rebalancing problem
    - 

# Lecture 4 – Tolerant Retrieval & Spelling Correction

## Wildcard queries

* Mon\* 🡪 find all docs containing any term beginning with mon
  + Easy with tree dictionary 🡪 retrieve all terms t in range mon <= t < moo
* \*mon 🡪 find all docs containing any term ending with mon
  + Duplicate tree, reverse terms (factor 2 memory consumption)
  + Find terms in range nom <= t < non
* Result: Set of terms that are matches for wildcard query
* M\*nchen
  + Lookup m\* and \*nchen in both trees and intersect (Expensive)
  + Alternative: **permuterm index**

### Permuterm Index ???

* Rotate every wildcard query, so that the \* occurs at each possible position
* Store each of these rotations in an additonal dictionary which maps to the original term
* 
* Permuterm 🡪 term mapping
* 
* Queries
  + For X, look up X$ e.g. hello
  + For X\*, lookup $X\* e.g. hel\*
  + For \*X, lookup X$\* e.g. \*lo
  + For \*X\*, lookup X\* e.g. \*ell\*
  + For X\*Y, lookup Y$X\* e.g. hel\*o 🡪 o$hel\*
* Multiple wildcards in one query cannot easily be handled with this data structure (except case above)
* Rotate query wildcard to the right
* Use tre lookup as before
* Permuterm empirically more than **quadruples** the size of dictioanry compared to regular tree.

### k-gram Indexes

* More space efficient than permuterm index
* Enumerate all character k-grams (sequence of k characters) occuring in a term
* 2-grams = bigrams, 3-grams = trigrams
* April is the cruelest month 🡪 bigrams:
  + $a ap pr ri il l$
  + $i is s$
  + $t th he e$
  + $c cr ru ue el le es st t$
  + $m mo on nt h$
* $ special word boundary symbol
* Maintain inverted index from bigrams to terms that contain that bigram
* 
* With this approach: 2 different types of indexes
  + Term-document inverted index for finding documents based on a query consisting of terms
  + K-gram index for finding terms based on a query consisting of k-grams
* Query mon\* can now be run as:
  + $m AND mo AND on
  + Returns all terms with the prefix mon
  + And many false positives 🡪 MOON 🡪 AND-operator has no order
  + postfilter these terms against query
  + Lookup result of filter in term-document inverted index

**K-gram index is more space efficient, but post filtering**

**Permutation index more space but no postfiltering.**

**Why doesn’t Google fully support wildcard queries? 🡪 Expensive**

### Processing wildcard queries in the term-doccument index

* Problem 1
  + Potentially large number of Boolean queries
  + Most straightforward semantics: Conjunction of disjunctions
  + Very expensive
* Problem 2
  + Users hate to type
  + If abbreviated queries are allowed, users will use them a lot!

## Edit distance

Two use cases:

* Correcting documents being indexed
* Correcting user queries

Two different methods:

* Isolated word spelling correction
  + Check each word on its own for misspelling
  + Will not catch typos resulting in correctly spelled words:
    - An asteroid that fell form the sky
* Context-sensitive spelling correction
  + Look at surrounding words
  + Can correct form/from error from above

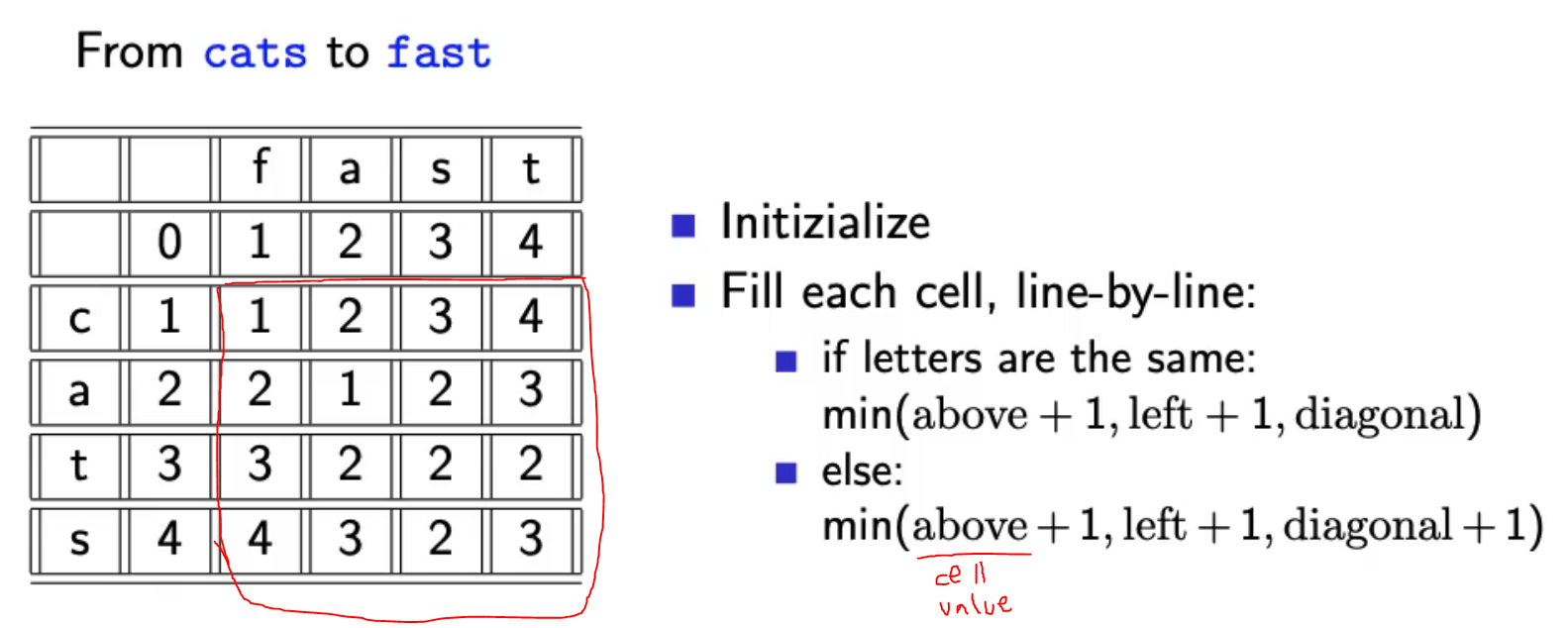
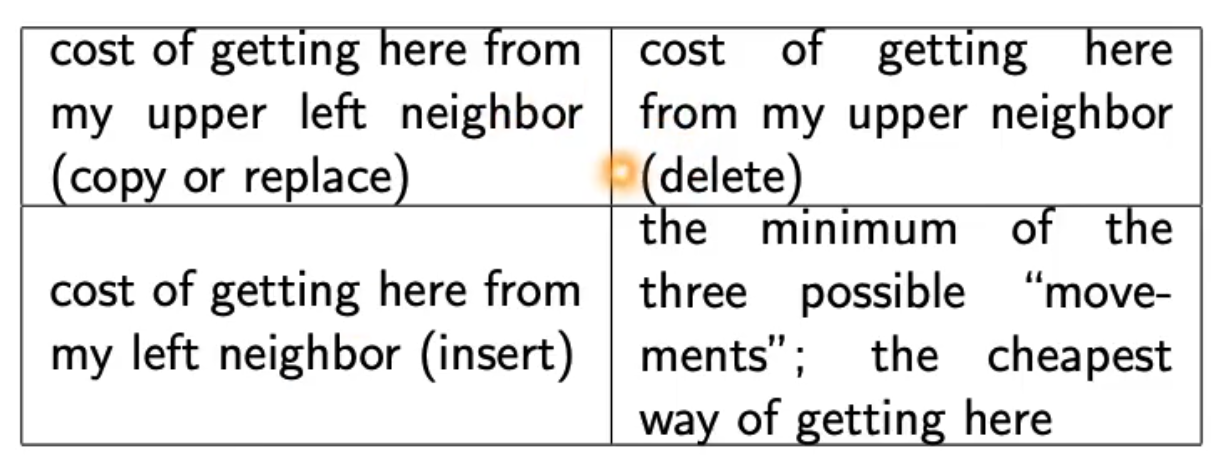
### Correcting documents

* In IR for errors in documents, e.g. OCR character recognition.
* General philosophy in IR: don’t change the documents.

### Correcting queries

* Premise 1: There is a list of “correct words” from which the correct spellings come from.
  + Dictionary
  + Industry-specific dictionary
  + Term vocabulary of the collection, appropriately weighted (remove very infrequent words)
* Premise 2: We have a way of computing the distance between a misspelled word and a correct word.
* Simple spelling correction algorithm: return correct word with smallest distance to misspelled word.

### Edit distance

* The edit distance between s1 and s2 is the minimum number if basic operations that convert s1 to s2.
* Levenshtein distance:
  + Basic operations: insert, delete and replace
  + dog do 🡪 1 (insert)
  + cat cart 🡪 1 (insert)
  + cat cut 🡪 1 (replace)
  + cat act 🡪 2 (insert, delete)
* Damerau-Levensthein distance (+ operation transposition = exchange) :
  + cat act 🡪 1 (transposition operation)
* How to compute distance 🡪 Dynamic Programming problem
* 
* Bottom right corner is minimum Levenshtein distance = 3
* 
* If we need to backtrack we can write it like this 😊
* 

### Weighted edit distance

* Weight of an operation depends on the characters involved
* 
* M more likely to be mistypes as n than q
* Replacing m by n is smaller edit distance than replacing m by q!
* We require a weight matrix as input 🡪 Modify algorithm to handle weights

### Using edit distance for spelling correction

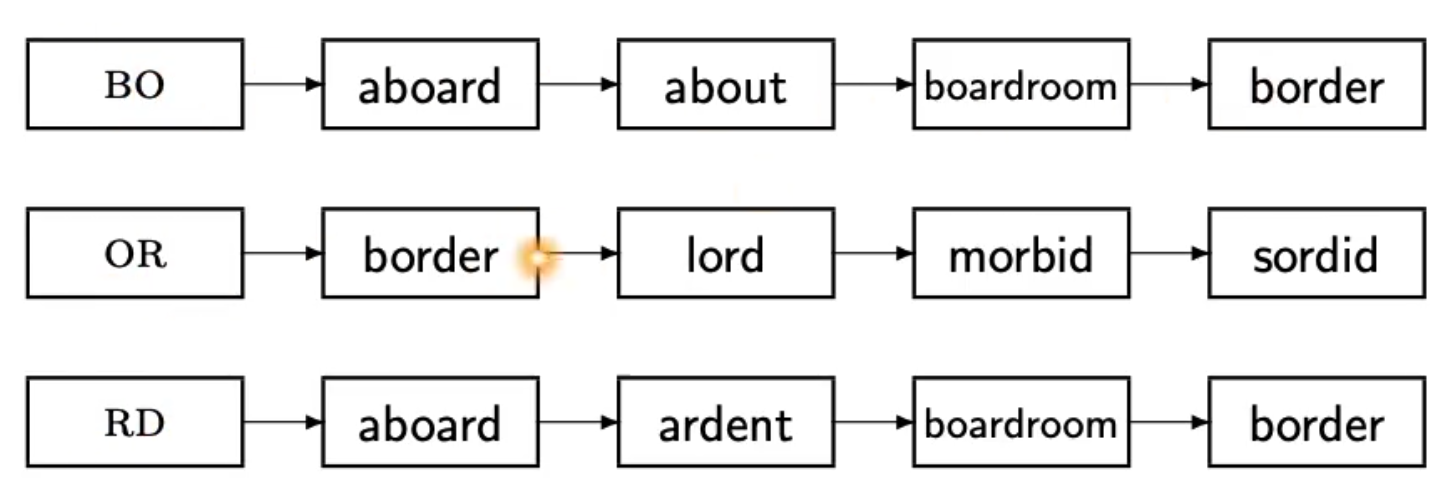
* Comparing query to all dictionary terms is very expensive!
* Approach 1:
  + Compare to all terms, but stop after specific edit distance (avoid unnecessary computations)
* Approach 2:
  + Generate all writing variations of query and compare to dictionary
* Approach 3:
  + Add some writing variations of the dictionary to the dicitionary 🡪 expensive but precalculated, increases dictionary size

### Example Levenshtein OSLO-SNOW



## Spelling correction

### k-gram indexes for spelling correction

* Enumerate all k-grams in the query term
* Example: bigram index\_ misspelled word bordroom
* Bigrams: bo, or, rd, dr, ro, oo, om
* Use the k-gram index to retrieve correct words that match query term k-grams
* Threshold by number of matching k-grams
* E.g., only vocabulary terms that differ by at most 3 k-grams
* 
* Apply intersection algorithm which contain all bigrams 🡪 boardroom
* Fixed number of k-grams that differ does not work for words of differing length!
  + NOVEMBER
  + Trigrams: nov, ove, vem, emb, mbe, ber
  + Query term DECEMBER:
    - Matches: emb, mbe, ber
  + 3 trigrams overlap out of 6 for each word, 9 different overall.
* How can we turn this into a normalized measure of overlap?

### Jaccard coefficient

* Measure overlap of two sets
* Let A and B be two sets
* Jaccard coefficient:
  + How many trigrams occur in both terms
  + Overall number of trigrams
  + 
* If A=B 🡪 1 perfect
* If no overlap 🡪 0 no match
* Always assings value between 0 and 1.
* DECEMBER NOVEMBER 🡪 3/9 🡪 1/3
* Application to spelling correction: declare match if the coefficient is > 0.8

### Context-sensitive spelling correction

* „an asteroid that fell form the sky“
* Hit-based spelling correction
  + Retrieve correct terms close to each query term
  + For „flew form munich“
    - Flea for flew
    - From for form
    - Munch for munich
  + Now try all possible resulting phrases as queries with one word „fixed“ at a time
    - Flea form munich
    - Flew from munich
    - Flew form munch
  + The correct query „flew from munich“ has the most hits
  + Not very efficient, combinatorial explosion

#### General issues in spelling correction

* User interface
  + Show user results of spelling correction and let user decide (can be overwhelming)
  + Automatic vs. Suggested correction
  + *„Did you mean“* only works for one suggestion
  + **Simple** vs. Powerful UI (Google)
* Cost
  + Spelling correction is potentially expensive
  + Avoid running on every query 🡪 maybe just on queries that have few matches
  + Guess: modern search engines use spelling corrections

# Lecture 5 – Index Construction

* Distributed index construiton: MapReduce
* Dynamic index construction: how to keep the index up-to-date as the collection changes
* Open source indexing library lucene

## Introduction

* Design decisions are based on hardware constraints

## Reuters RCV1 statistics

* 800k docuents
* 200 tokens per document
* 400k terms
* 6 bytes per token
* 7.5 bytes per term
* 100 Million non-positional postings
* Average frequency of a term
  + 400k / 800k\*200 = 0.0025
* How many positional postings?
  + 100 M \* 200 = 20 Gigabyte

## BSBI

* Sort based index construction
* As we build index, we parse docs one at a time
* The final postings for any term are incomplete until the end.
* Can we keep all postings in memory and then do the sort in-memory at the end?
* Disk access is not allowed.

## External sorting algorithm

* Sort 100M non-positional postings
  + Each posting has 12 bytes
* Define block of 10M such postings
  + We can easily fit that many postings into memory
  + Blocks of term id/document id pairs
  + For each block
    - Accumulate postings
    - Todo
    - Todo
* Problem: keep entire dictionary in memory in order to implement term 🡪 termID mapping.

## Single-pass in-memory indexing (SPIMI)

* Generate separate dictionaries for each block – no need for termID mapping
* Don’t sort across blocks/ term-DocId paris, accumulate postings in postings lists as they occur
* 1 index per block, then combine
* ??? why don’t sort?

## Distributed indexing

* Uses a distributed computer cluster
* Individual machines are fault-prone
* How to exploit such a cluster?
* Maintain master machine directing the indexing job – considered „safe“
* Break up indexing into sets of parllel tasks
* Master machine assigns each task to idle machine
* Two sets of tasks
  + Build subsolutions (Parsers)
    - Parser reads a document at a time and emits (term, docId) pairs
    - Parser writes pairs into j term-partitions
    - TODO
  + Merge subsolutions (Inverters)
    - Sees terms partitions
    - Collects (term,docID) pairs 🡪 takes all (a-f) partitions from all Parsers
    - Sorts and wirtes to postings lists
    - Data flow (ugly image) TODO
* Break the document collection into splits (block) = subset of documents

## Dynamic Indexing

* Dynamic collections that change over time
* Docs
  + Insert
  + Delete
  + Modify
* Dynamically modify
  + Dictionary (add /remove terms)
  + Add/remove elements from a postings list
* Naive approach:
  + Rebuild index from scratch from time to time (if small than its actually easy)
* Simple approach:
  + Maintain big main index on disk
  + New docs go into small auxiliary index in memory
  + Search accross both and merge results
  + Periodically merge auxiliary index into big index
  + Deletions:
    - Invalidation bit-vector for deleted docs (flagged)
    - If flag is set, remove from results shown to user
  + Issues:
    - Frequent merges
    - Poor search performance during index merge (disk seeks)
* Logarithmic merge
  + Series of indexes, increase by factor 2
  + Keep smallest Z0 in memory
  + Larger ones on disk
  + If Z0 gets too big (>n), write to disk as IO
  + TODO…… copy table from lecture video

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Z0 | I0 | I1 |  |  |
| 50 |  |  |  |  |
| 50 | 50 |  |  |  |
| 50 |  | 100 |  |  |
| 1 | 50 | 100 |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

## Lucene

* Open source Java library
* Lets you add search to your application
* Not a complete search system by itself
* Used by: wittre, LinkedIn, Reddit, CiteSeer
* Ports integration to other languages