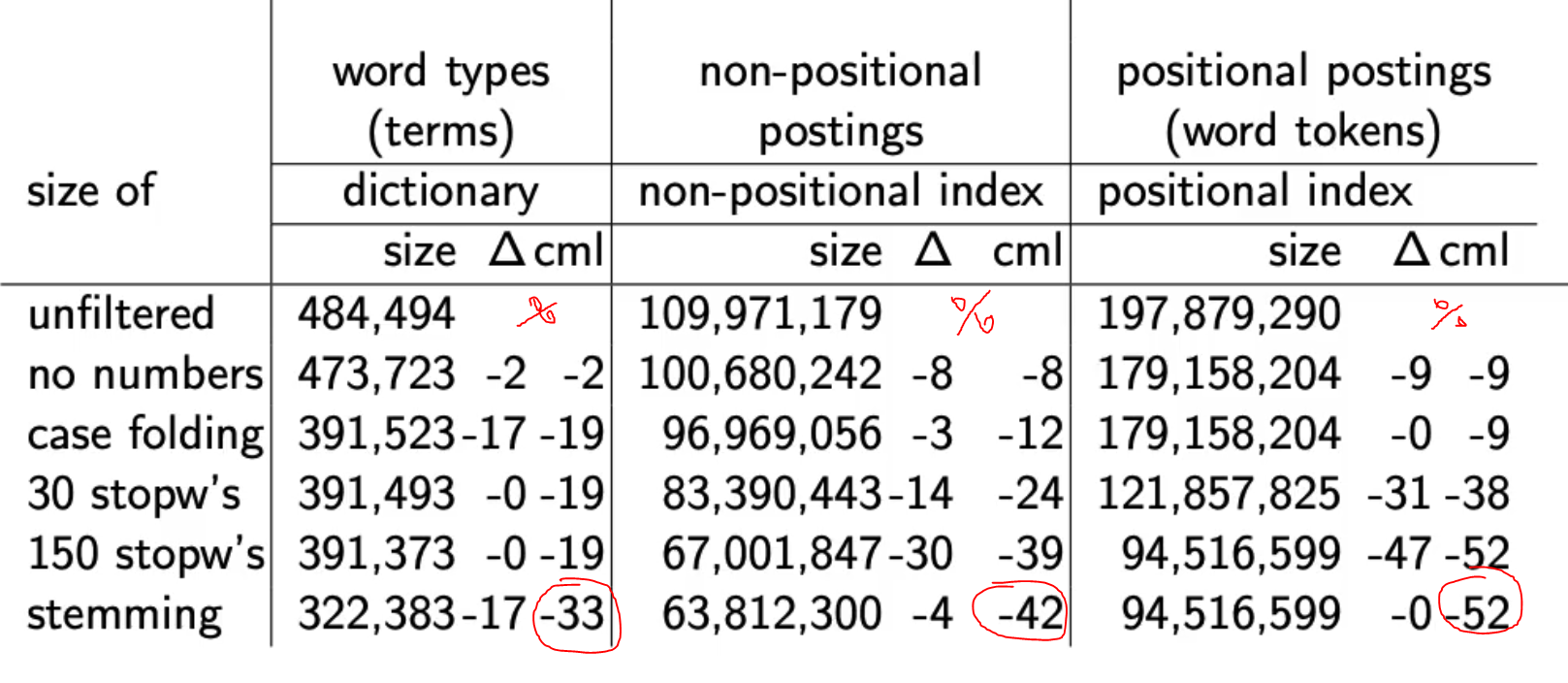
# 09.11.2021 – Compression

* Motivation
  + Less disk space
  + Keep more stuff in RAM
  + Increase speed of transferring data from disk to RAM
    - Read compressed data and decompress in memory is faster than reading uncompressed data (if decompression algos are fast)
* Lossy vs. Losless compression
  + Lossy: discard compression (e.g. MP3, JPEG)
    - Preprocessing 🡪 lossy compression (downcasing, stop words, stemming, number elim)
* How can we compress the dictionary of the inverted index?
* How can we compress the postings of the inverted index?
* Term statistics: how are terms distributed in document collections and how can we estimate this number?

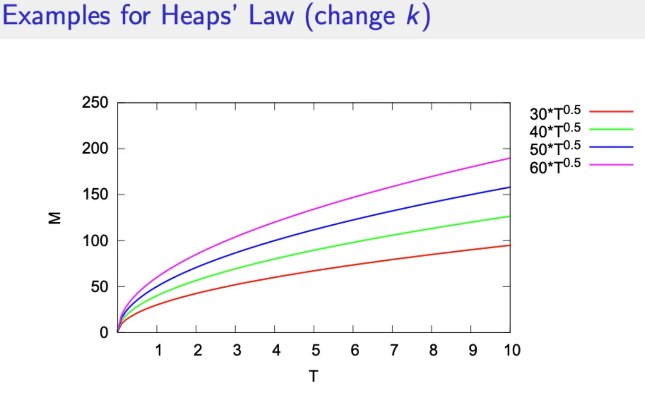
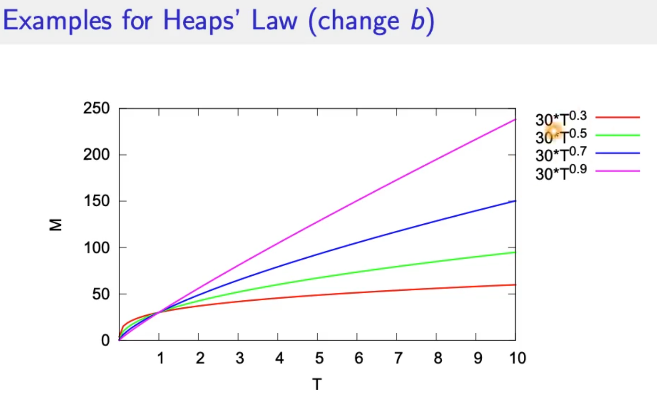
## Term statistics

Lossy compression by preprocessing:

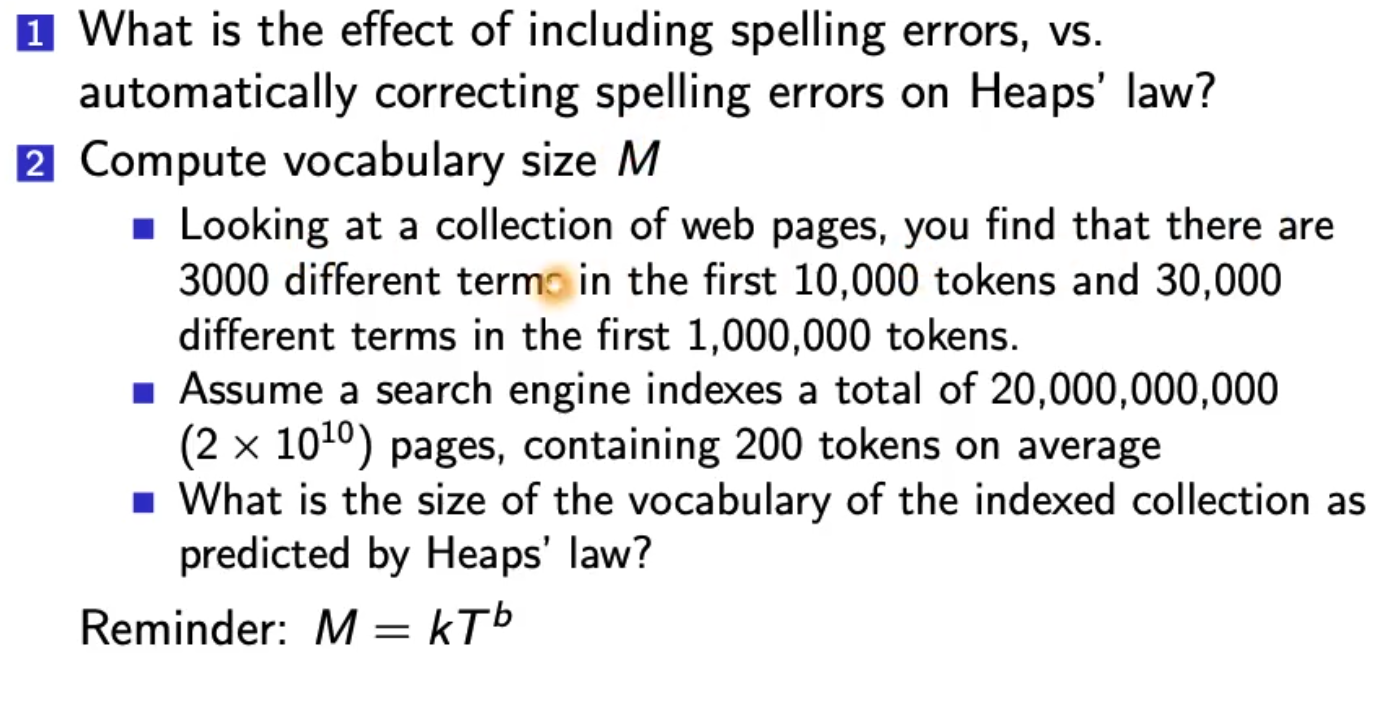


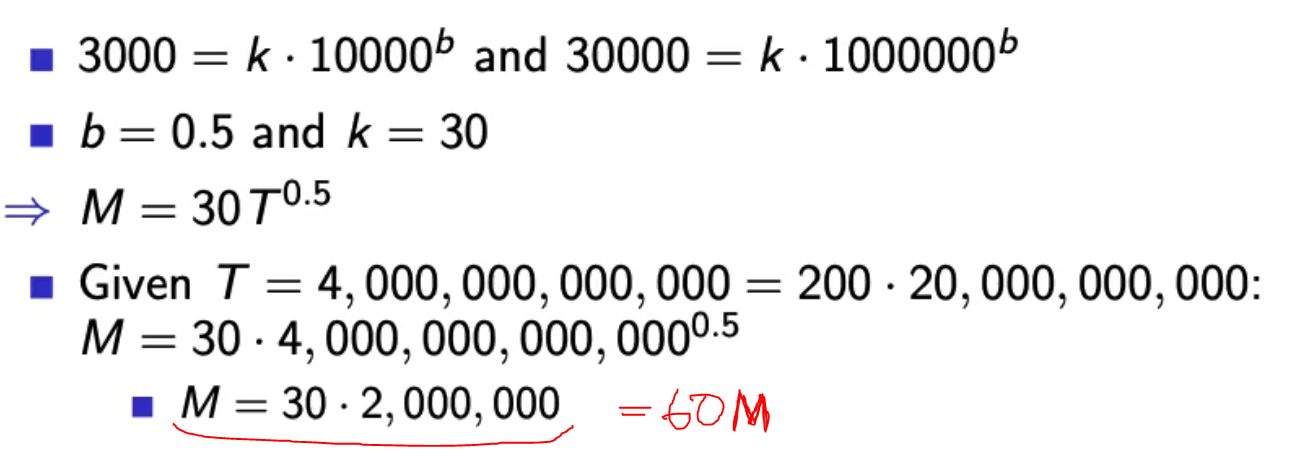
## Heap’s law: Can we assume there is an upper bound?

* Not really: many long words, very infrequent
* The vocabulary will keep growing with collection size, but how much?
* Heap’s law: M= kTb
  + M is the size of the vocabulary
  + T is number of tokens in collection
  + Typical values for k and b are:
    - 30 <= k <= 100
    - b ~ 0.5

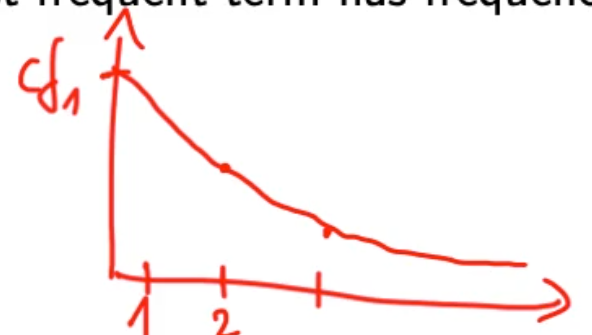


## Exercise

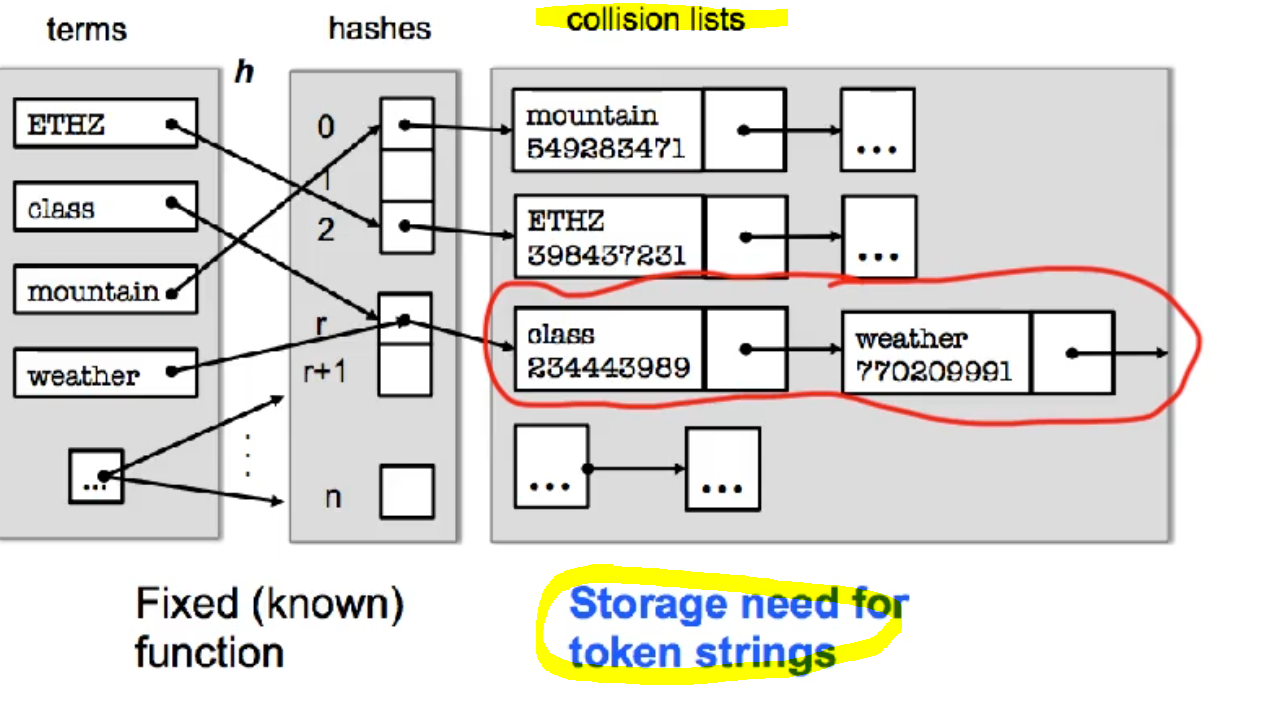
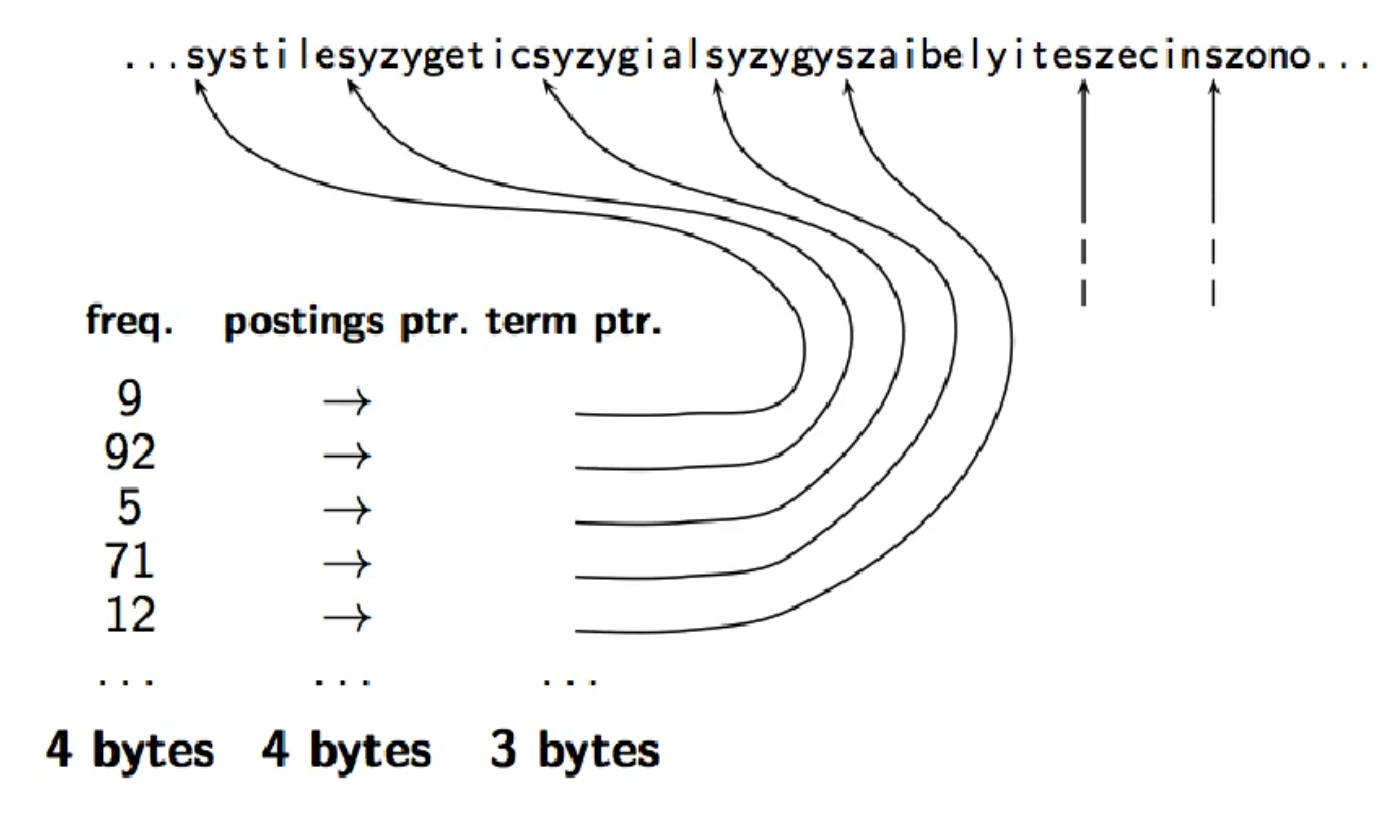
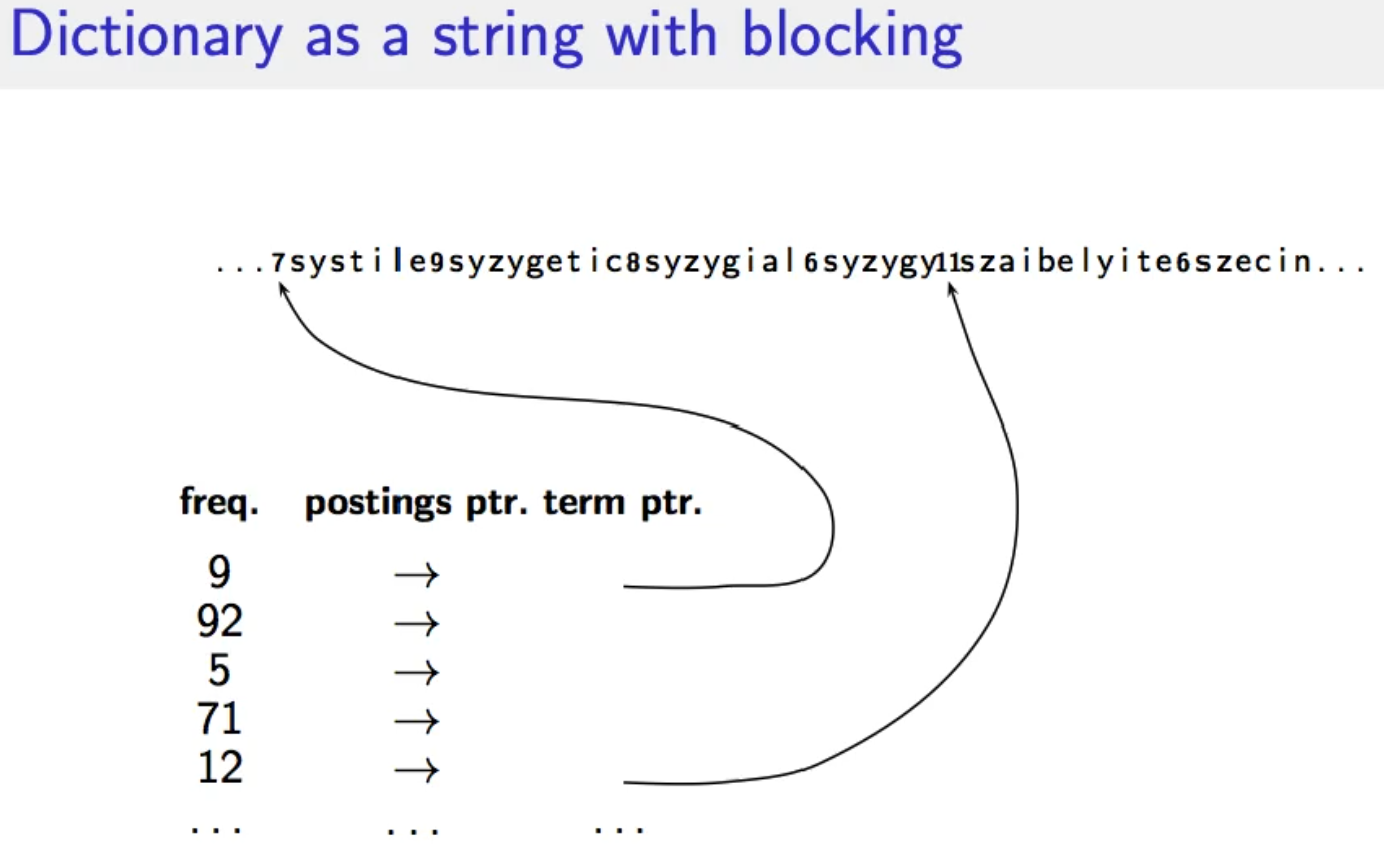
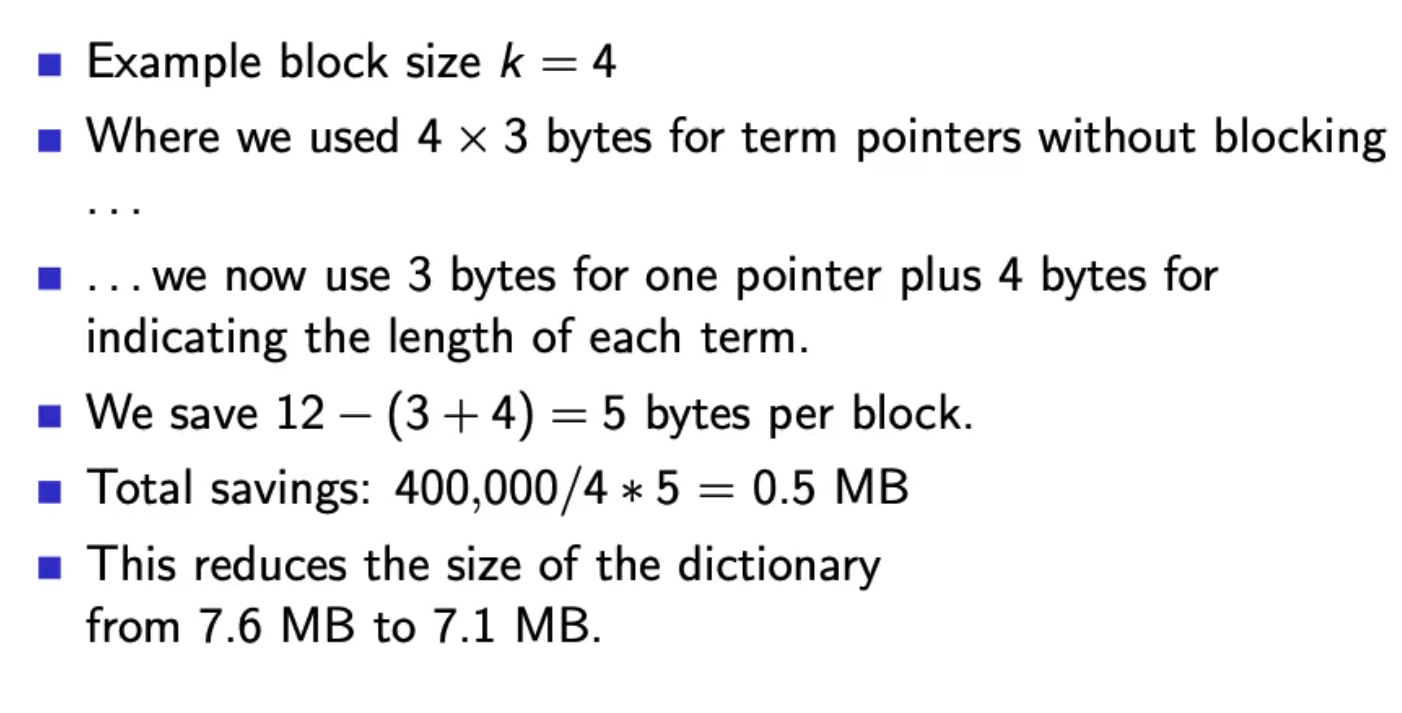
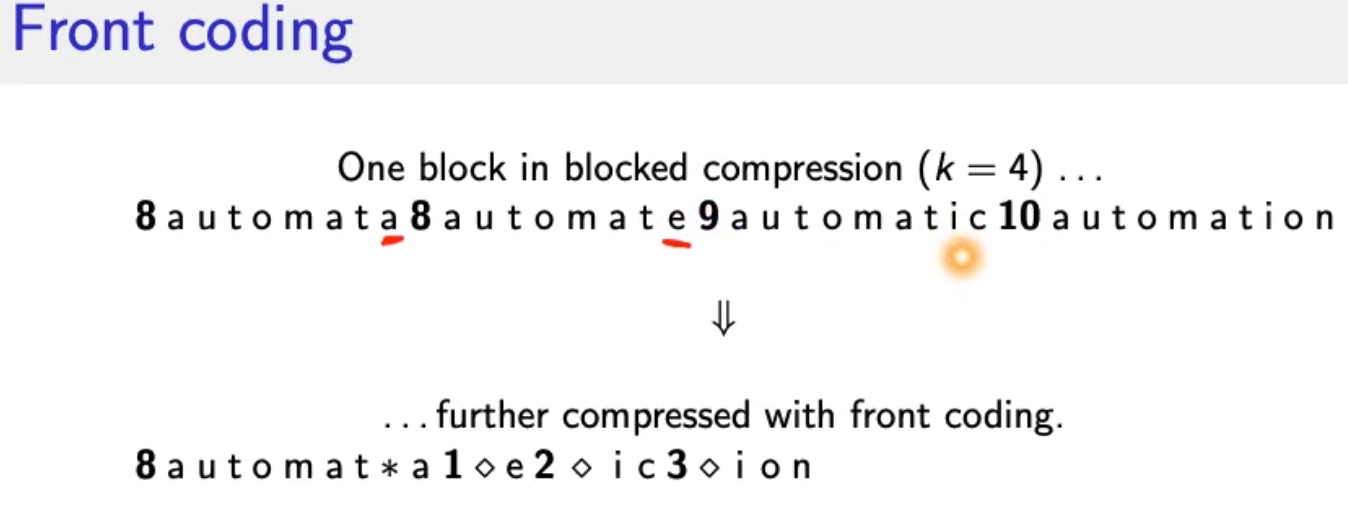
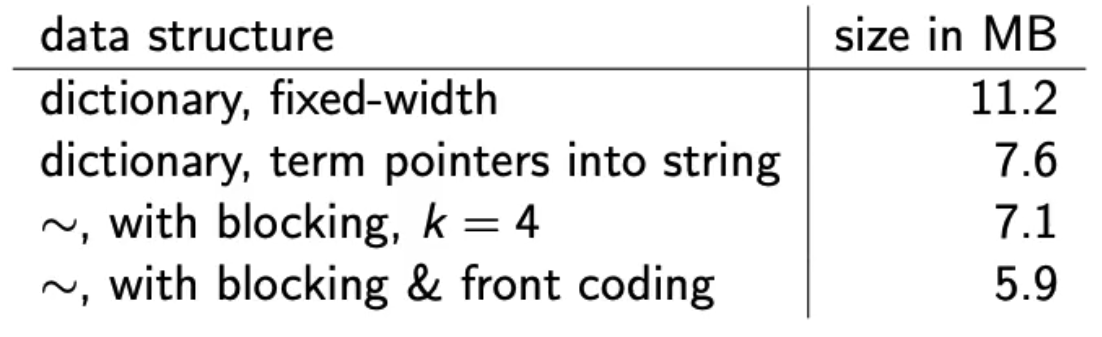




Zipf’s law

* Heaps law: growth of vocabulary
* Zipfs law: frequent vs. infrequent terms
* In natural language, there are a few very frequent termsn and very many very rare terms
* Zipf’s law
  + Ith most frequent term has frequency cfi proportional to 1/i
  + 
* If the most frequent term (the) occcurs cf1 times, then the second most frequent term (of) has frequency cf2 = cf1/2

## Dictionary compression

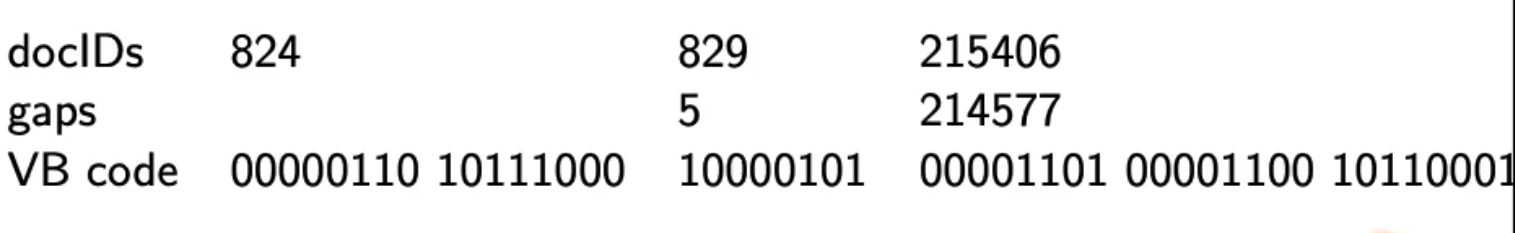
* Dictionary is small compared to postings file
* We want to keep it in RAM
* 
* Dictionary fixed-width entries are bad!
  + Supercalifragilisticexpialidocious
  + Average length of term in english 🡪 8 characters
  + 
  + Where does the string know how to stop???
  + 
  + 4 terms point to the same block 🡪 Following a pointer into a block, who do I know which term to read?
  + 
  + 
  + 

## Postings compression

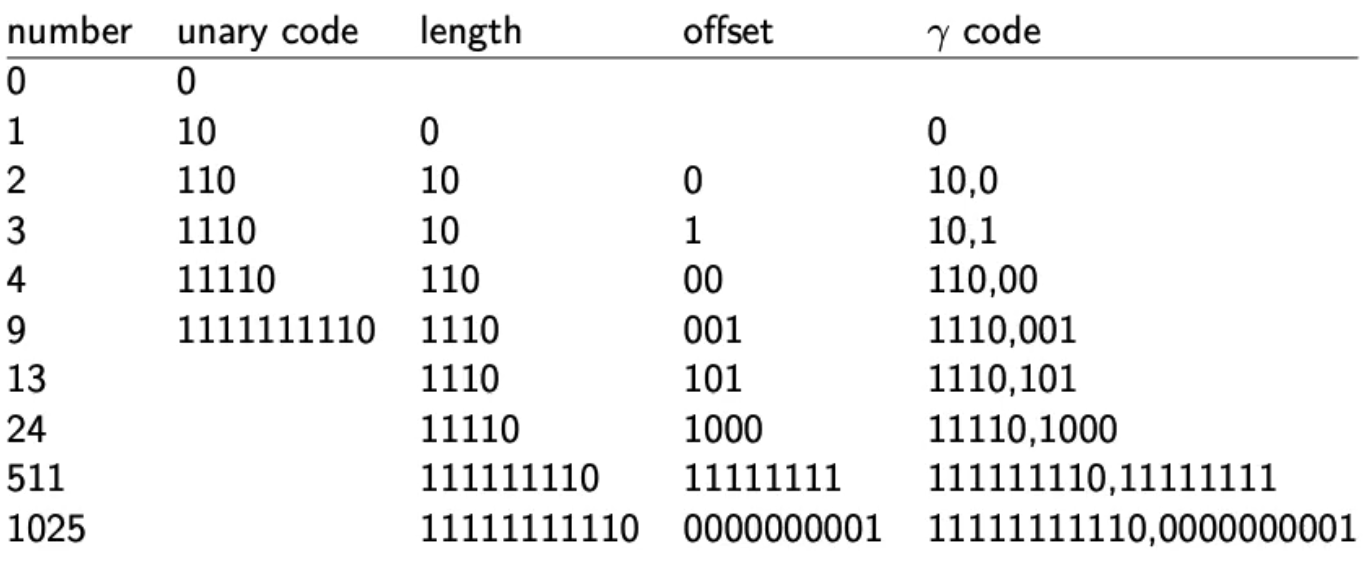
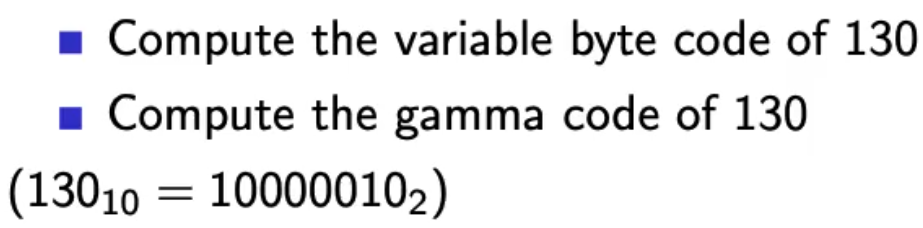
* Postings file is much larger than dictionary (at least factor 10)
* Key idea: store each posting compactly
* Each postings list is ordered in increasing order of docID
  + Computer : **283153, 283159, 283202**
  + It suffices to store the gaps
    - **Computer: 283154, 5, 43**
* Gaps are small for frequent term, its never worse than storing the docID
* Variable length encoding:
  + For arachnocentric and other rare terms we will use about 20 bits per gap
  + For the and other very frequent terms we will use only a few bits per gap

### Variable Byte (VB) code

Key idea:

* Dedicate 1 bit (high bit) to be a continuation bit c
  + 0000 0000
* If the gap G fits within 7 bits:
  + Binary encide it in the 7 bits and set c = 1
  + 127 = 1111 1111
* If G doesn’t fit in 7 bits:
  + Set c = 0
  + Encode lower-order 7 bits and then use one ore more additional bytes
  + 
  + 128 = 0000 0001 1000 0000
  + The final byte has the c=1
* 

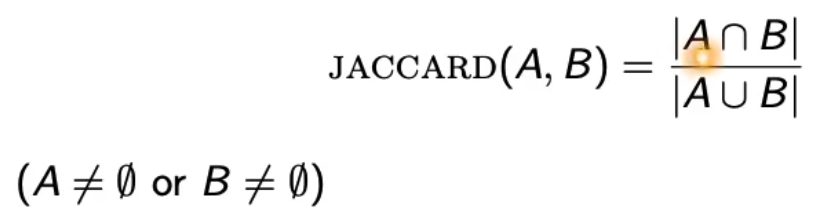
## Gamma codes for gap encoding

* Bitlevel code
* Gamme code is the best known of these bitlevel codes
* Introduce unary code 🡪 one symbol
* Represent 5 🡪 3 = 1110 with final 0
* Represent a gap G as pair of (length, offset)
  + Offset is gap in binary with leading bit chopped off (by convention, always!)
  + 13 🡪 1101 🡪 101 offset
  + Length is the the length of the offset
  + For 13, offset (101) the length is 3
  + Encode length in unary code 🡪 1110
  + Gamma code:
    - Concatentation of length and offset
    - 1110 101 🡪 1110 101
* 
* 
* VB 130 = 0000 0001 1000 0010
* Gamma 130:
  + Offset 000 0010
  + Length: 11111110
  + Gamma Code: 11111110 0000010
* 
* Gamma codes are always of odd length and within a factor of 2 of the optimal length encoding.
* Gamma code is prefix free! We can do intersection algorithm on bit sequences
* Gamma code is parameter free, universal and doesn’t care about the distributions of gap values
* Machines have word boundaries 8, 16, 32 bit
* Compressing and manipulating at granulartiy of bits can be slow
* Variable byte encoding is aligned and thus potentially more efficient, and conceptually simpler

# 11.11.2021 – Scoring

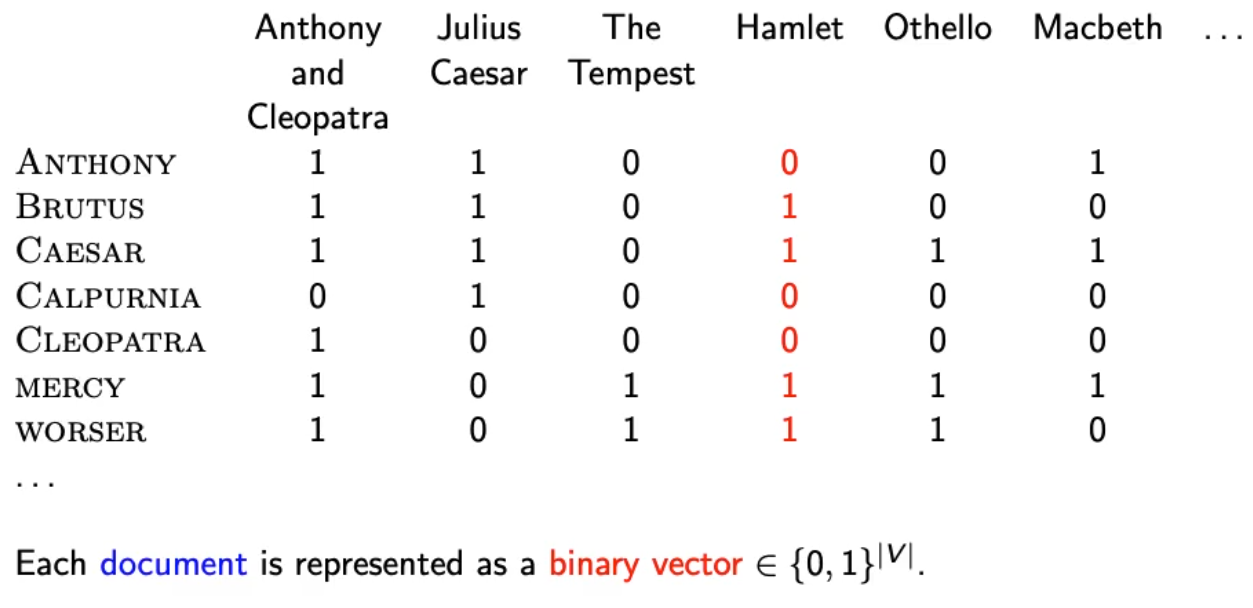
* Ranking: why it is important
  + Boolean
    - match or no match, good for expert users and applications
    - not good for the majority of users
    - gives very few = 0 or too many results (1000s)
  + Ranked retrieval
    - Large result sets are not an issue
    - If most relevant is on top 🡪 😊
    - Just shop top 10 results 🡪 doesn’t overwhelm the user
    - More relevant results are ranked higherr than less relevant results.
* Term frequency: key ingredient for ranking
* Tf-idf ranking
* Vector space model

## Scoring

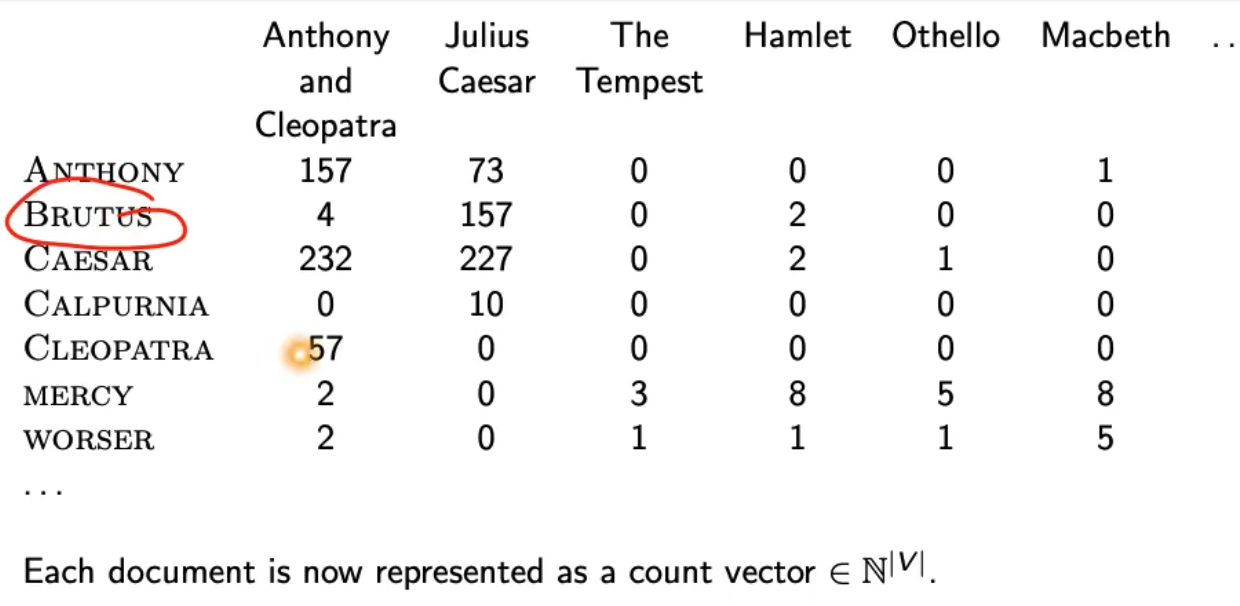
* Score = query-document pair 🡪 How well does a document fit to a query?
* One-term query
  + Query term does not occur 🡪 score = 0
  + The more frequent the query term in the document the higher the score
  + Query term once = x
  + Query term twice = not 2x, we don’t get double the information
* Jaccard coefficient
  + Measure of overlap of two sets
  + 
  + Jaccard(A,A) = 1
  + Jaccard(A,B) = 0 if A,B distinct
  + **Depends on size of the document 🡪 large document leads to low scores**
  + **Rare terms are more informative than frequent terms!**
  + **Does not consider term frequencies**
  + (Only exact term match) 🡪 can be fixed by proper normalization

## Term frequency normalization

Before: binary incidence matrix



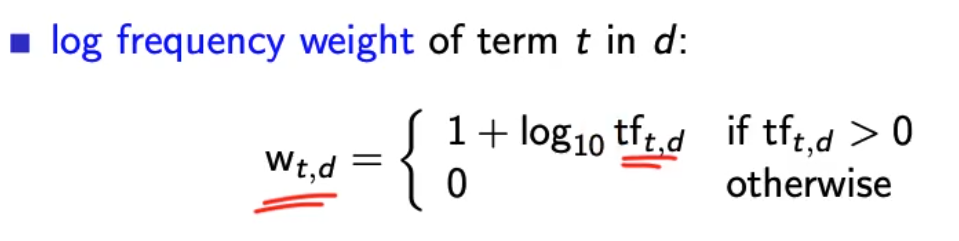
Now: Count matrix



## Bag of words model

* We do not consider the order of words in a document: same representation:
  + John is quicker than Mary
  + Mary is quicker than John
* This is a step back: positional index was able to distinguish these two documents.

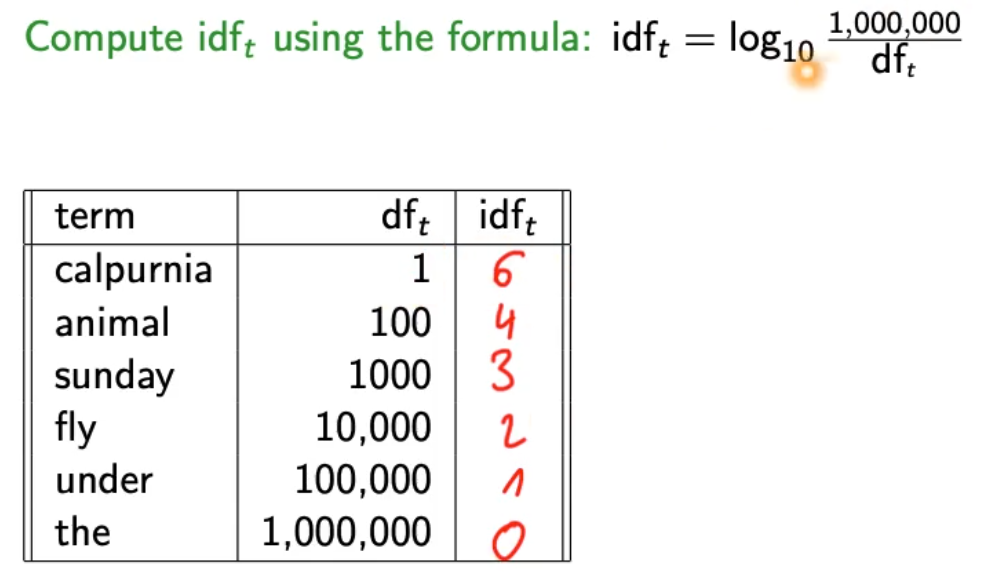
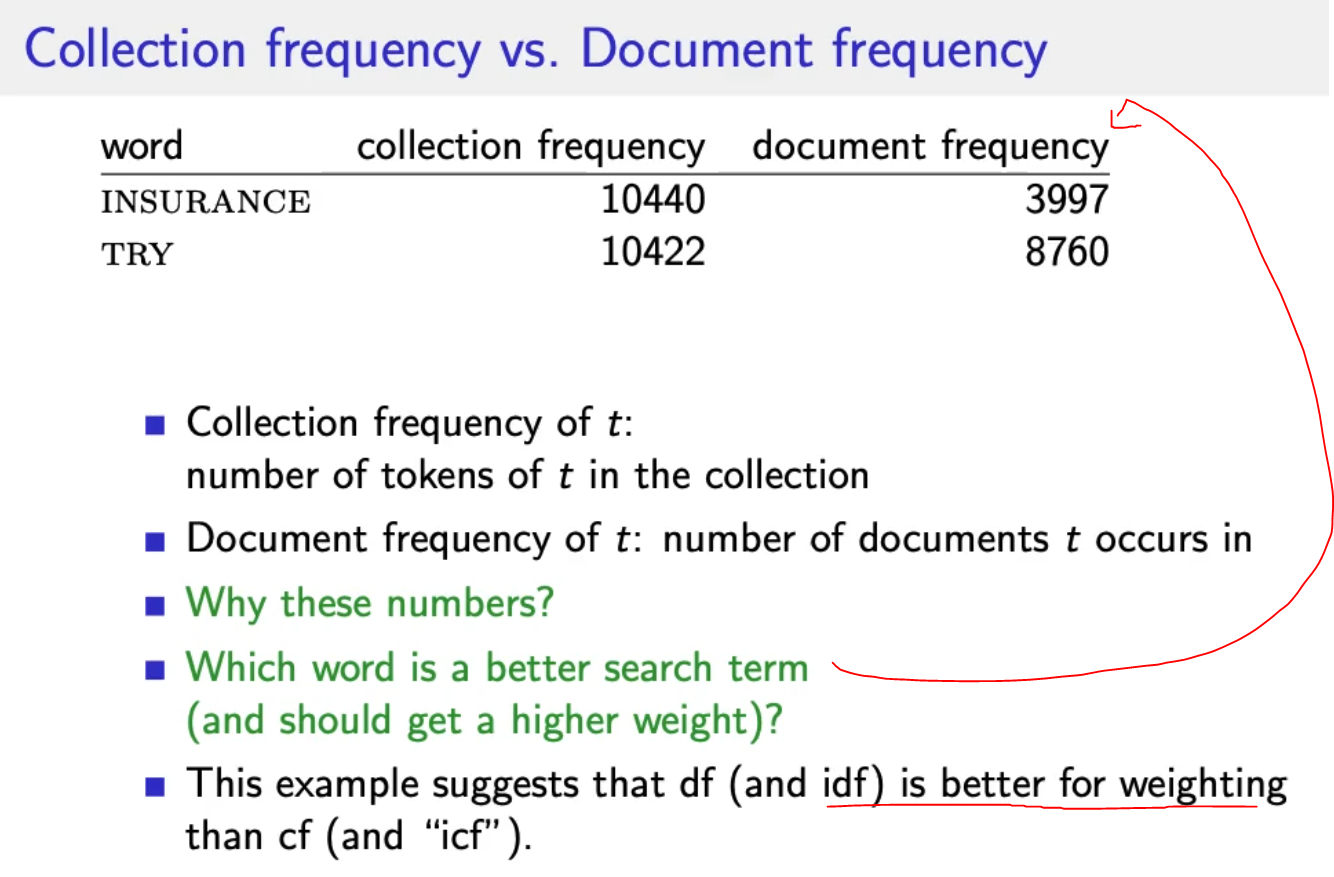
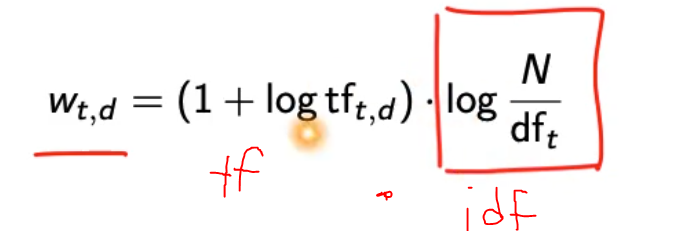
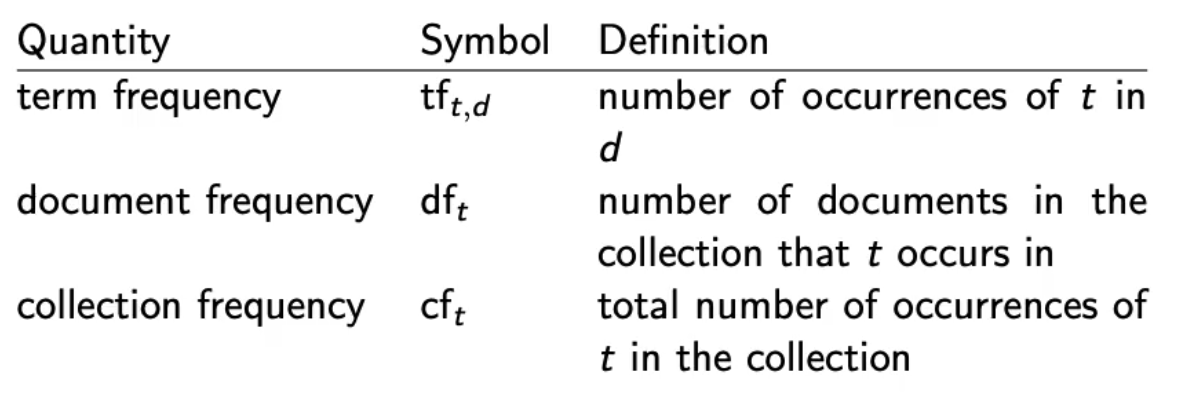
### Term frequency tf

* tf(term, document) 🡪 n = number of times the term occurs in this document
* Use tf when computing query-document score
* Raw term frequency is not what we want because
  + tf = 10 not 10 times more relevant than tf=1
* Instead: Log frequency weighting:
  + 
  + 0 🡪 0
  + 1 🡪 1
  + 2 🡪 1.3
  + 10 🡪 2
  + 1000 🡪 4

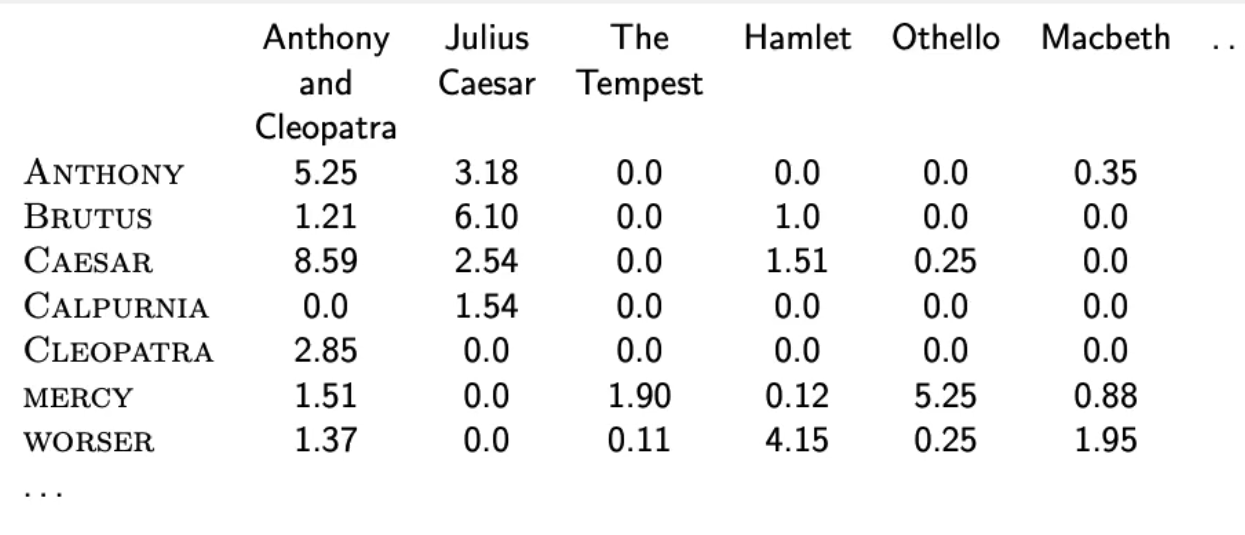
### Score for query-document pair

* Sum over terms in query and document the log frequency weighted term frequency.
* Score = 0 if none of the query terms is in the document.

## Tf-idf weighting

* Frequency in document vs. frequency in collection
  + Rare terms are more informative than frequent terms 🡪 high weight
  + Frequent terms are less informative than rare terms 🡪 low weight
* We will use the document frequency to factor this into computing the score
* Idf weight
  + Df is the document frequency df(t)
  + Global measure for entire collection
  + Df is inverse measure of informativeness
    - Idf weight
      * Idf(t) = log10 N/df(t)
    - N = number of documents in collection
  + Idf is a measure of informativeness
* 
* Logarithm of idf needs to fit the logarithm of tf such that these can “outweigh” each other
* **Idf affects the ranking of documents for queries with at least two terms**
  + Changes absolute numbers but not the ranking
* For example “arachnotcentric line”
  + Idf will increase arachnocentric importance score
  + Idf will decrease line importance score
* 
* Tf-idf is the product of tf weight and idf weight
  + 
  + Increases if term occurs multiple times in document
  + Increases if term is rare
* 

## The vector space model

* tf-idf matrix
  + 
* Documents as vectors e.g. tf-idf vector (high dimensional 🡪 number of words in vocabulary)
* Terms are axes of the space
* Documents are points or vectors in this space
* Very high-dimensional – 10M+ dimensions when apply this to web search engines
* Each vector is VERY sparse – most entries are zero

### Queries as vectors

* Also represent queries as vectors in this high dimensional space
* Rank documents according to proximity to the query
* Proximity = simiarity ~ negative distance
* Rank relevant docs higher than non-relevant (get away from boolean model)

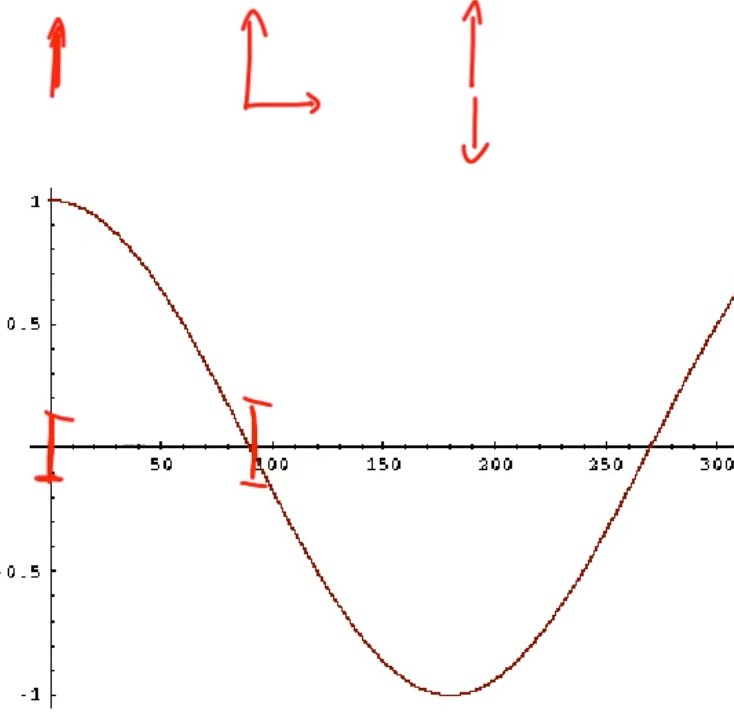
### Euclidian distance?

* + Documents have words often
  + Query is short
  + Bad idea 🡪 large for vectors of different lengths!

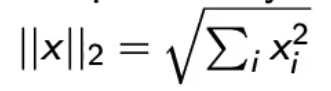
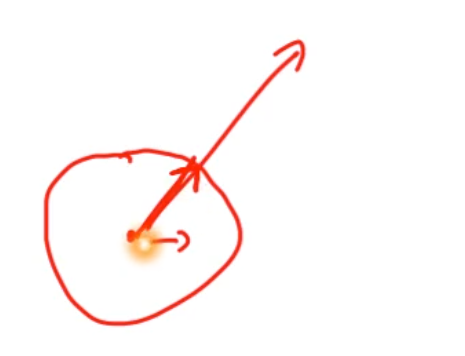
### Use angle

* + Rank documents according to angle with query
  + Thought experiment:
    - Document d1, append it to itself 🡪 d1.concat(d1) = d2
    - d1 and d2 semantically have the same content
    - Angle between d1 and d2 is zero = maximal similarity
    - Euclidian distance can be large

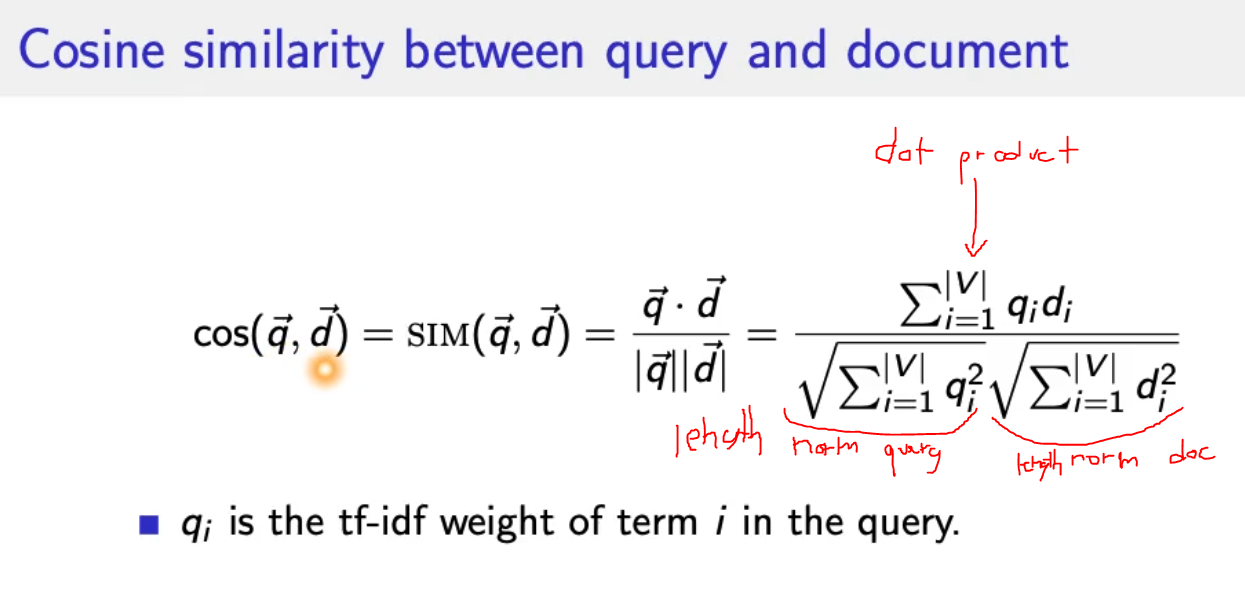
### From angles to cosine similarity

* It’s the same to rank documents with cosine(query, document) in increasing order
* 
* Value from 0 to 1

#### Length normalization

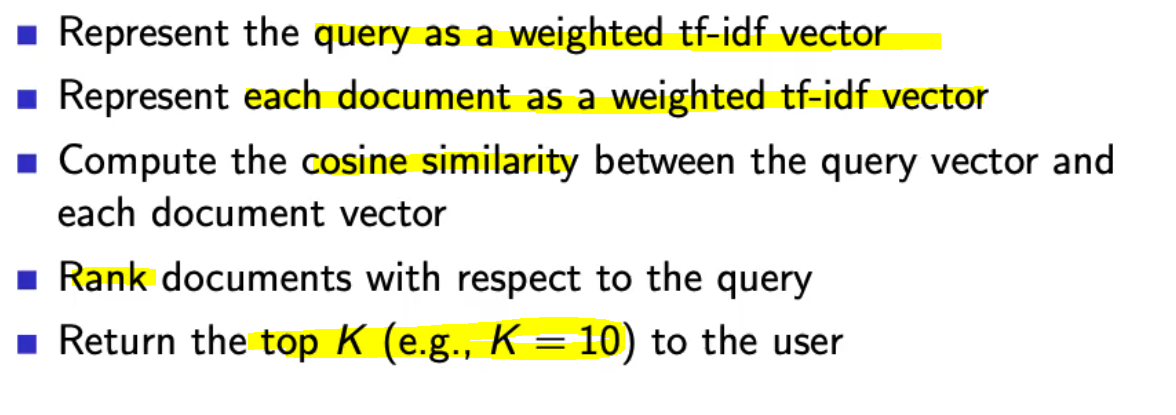
* Needed to compute the cosine
* L2 norm: 
* Maps vectors to multidimensional unit sphere
* 
* Longer and documents have weights of the same order of magnitude
* d1 and d2 from example above result in identical vectors

#### Cosine similarity between query and document



* is cosine similarity between q and d 🡪 cosine(angle between q and d)

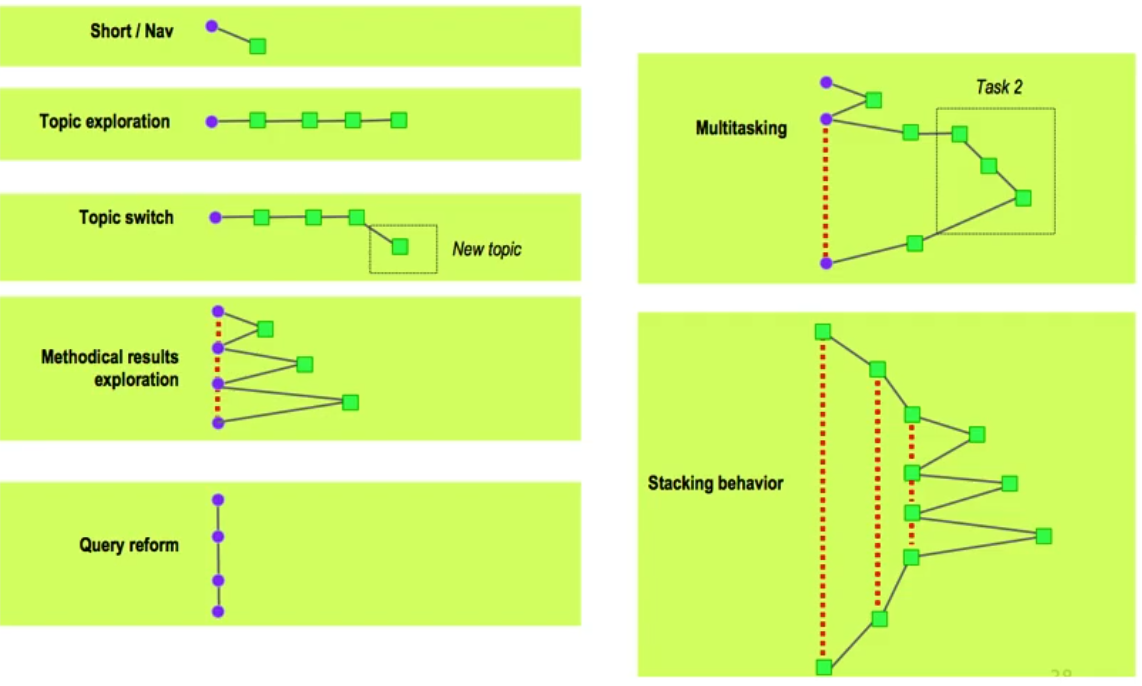
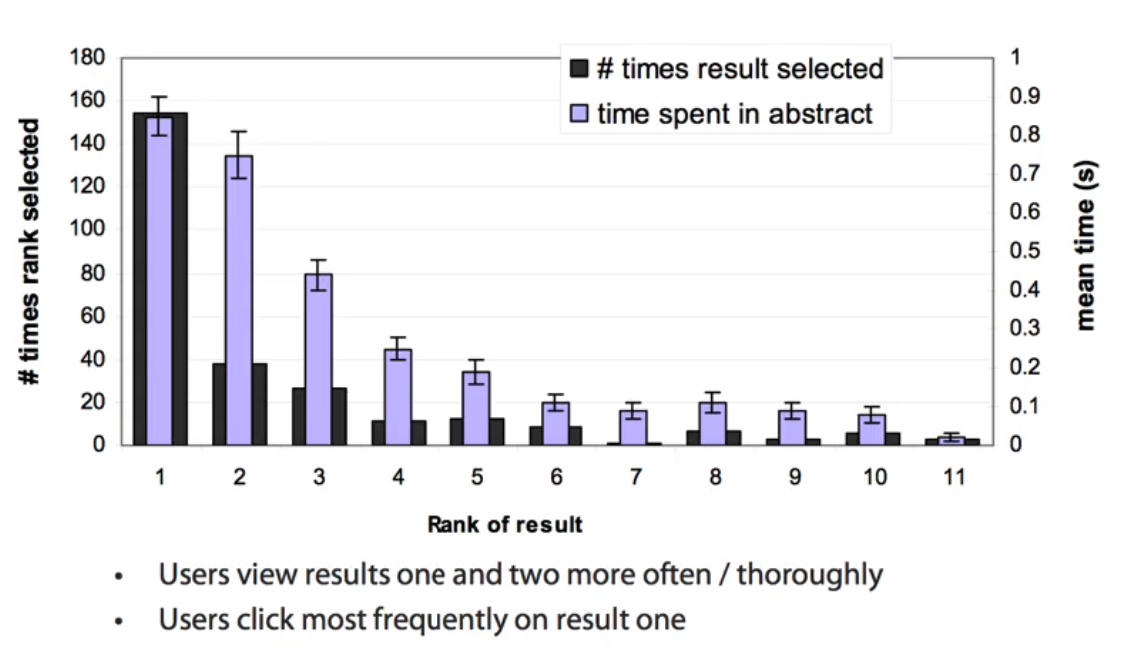
## Summary



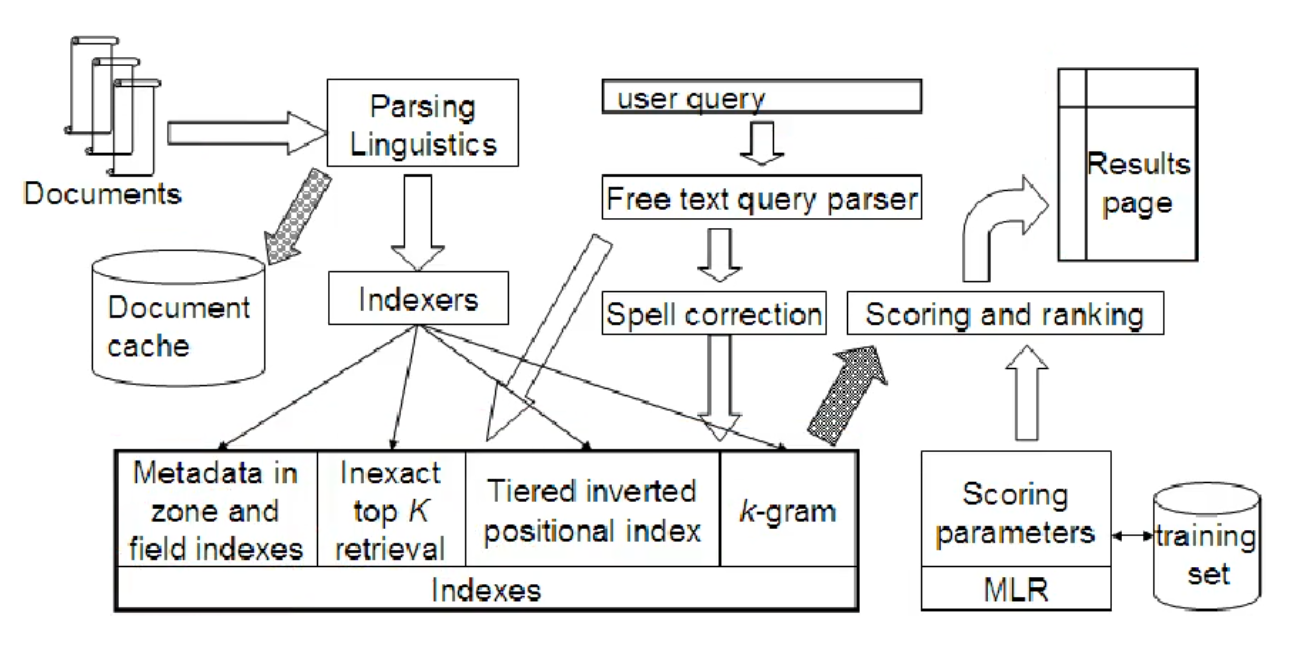
* for first point: just take idf vector from collection, no updates

# Lecture 08 – Ranking, Complete System, Summaries

## How can we measure how important ranking is?

* Interviews: Person just looks at the price, not at the name of the site
* Eyetracking
* Behaviors
  + 
* How many links do people look at?
  + People only look at 1+2 and the 3 a little, rest is irrelevant. 🡪 clicking is even more skewed
  + 
* 🡪 Top ranked page is most important!

## Complete search system



## Tiered indexes

* Create several tiers of indexes corresponding to importance of indexing terms
* During query processing, start with highest-tier index
* Highest-tier index 🡪 titles of documents 🡪 small index
* If we find too few index 🡪 query next index
* Example: 2-Tier system
  + Tier 1: Index of titles
  + Tier 2: Index of the rest of documents

## Document Cache

* We need this for generating snippets (= dynamic summaries)

## Zone indexes

* Separate the indexes for different zones e.g.
  + Body
  + Special paragraphs
  + Italic
  + Anchor text
* Small indexes 🡪 fast queries

## Machine-learning based ranking functions

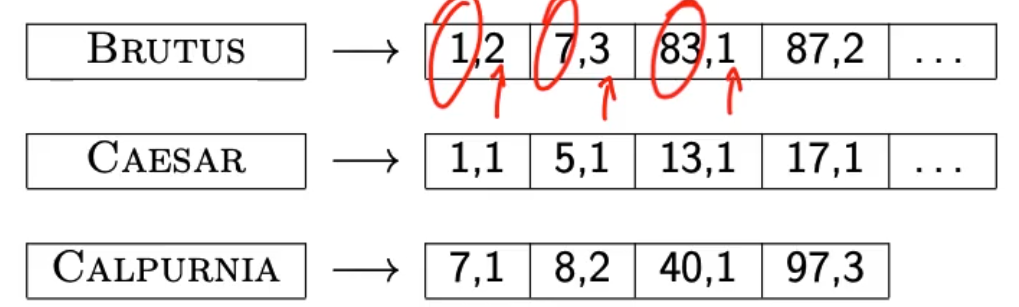
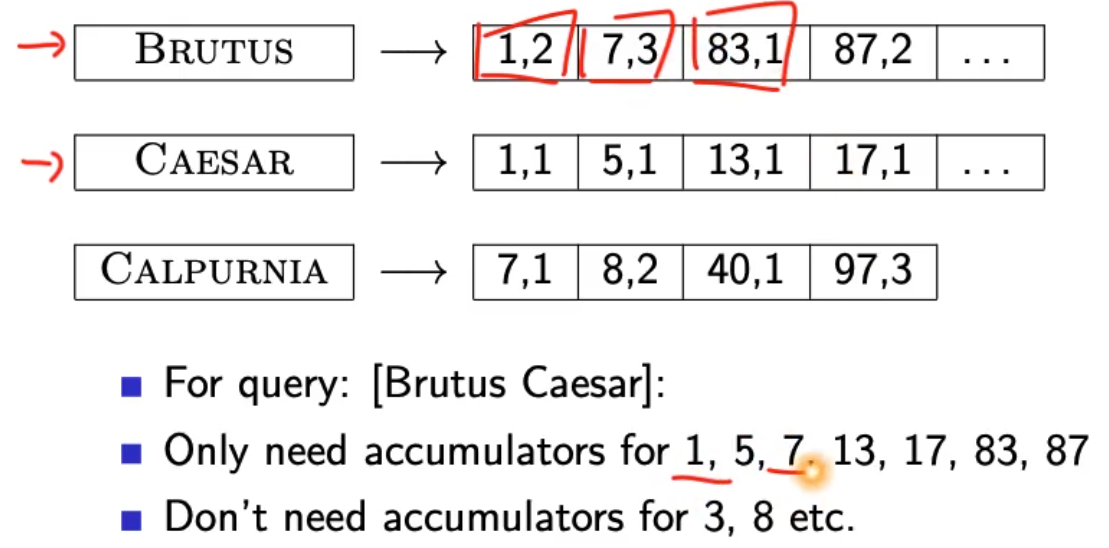
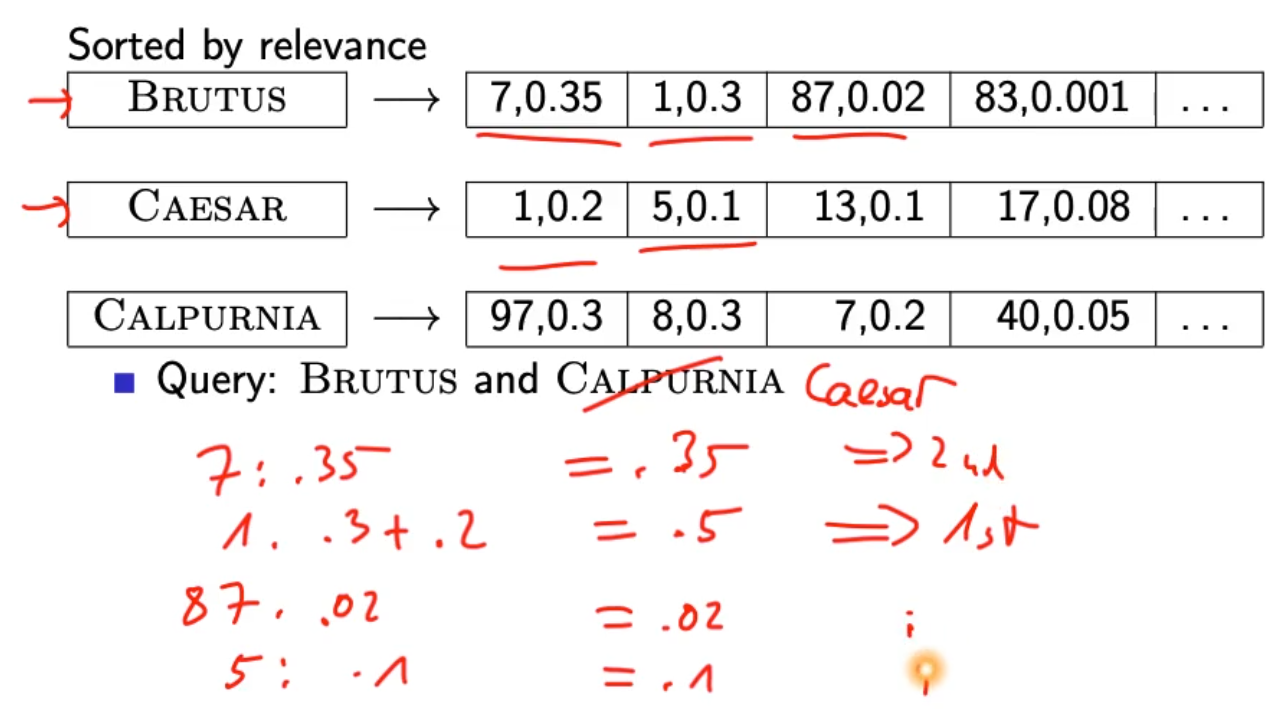
## Proximity ranking

* Rank documents in which the query terms occur in the same local windows higher than documents where query terms occur far from each other

## Query Parser

* IR systems often guess what the user intended
* London tower 🡪 „london tower“ bigram
* 100 Madison Av, New York 🡪 address 🡪 show map
* Parse query into:
  + Formal specification
  + Phrase operators
  + Proximity operators
  + Indexes

# Implementation of Vector space retrieval 🡪 Ranking

* **Term frequencies in the inverted-index:**
  + 
  + Also need positions
  + Store termfrequency(term,document) in addition to docID d in each posting
    - As integer frequency (easier to compress)
  + Additional space requirements are small.
* **How do we compute the top k in ranking?**
  + No need for complete ranking 🡪 best top k (e.g. k = 100)
  + Efficient way of computing top k? 🡪 Heuristic
* **Heuristic**
  + Idea: split into important and unimportant documents
  + Idea 1: **Reorder postings lists**
    - Instead of order by docID
    - Order by **expected relevance** (global)
    - Expected relevance could be 🡪 **PageRank** g(d) of page d 🡪 a measure of how many good pages hyperlink to d
    - Composite scroe: PageRank + cosine similarity to query
  + Idea 2: **Heuristics** to prune the search space
    - Not guaranteed to be correct, but rarely fails in practice
  + Document-at-a-time processing
    - docID/pageRank impose consistent ordering on postings list
    - We can intersect lists normally and compute similarity doc by doc
  + Weight-sorted postings lists
    - Order docs in posting list according to weight 🡪 normalized tf-idf weight
    - Documents in top k are likely to occur early in these ordered lists
      * Supports early termination, probability to change top k decreases with runtime
    - No consistent ordering of documents in postings list anymore
      * We no longer can do document-at-a-time anymore
  + Term-at-a-time processing
    - Completely process postings list of the first query term
    - Create **accumulator** for each docID you encounter
      * Only create while step through lists! (memory requirement)
      * 
      * Sum up scores from all terms to accumulators for a document
    - Completely process postings lists of the second query term
    - 

### Summary

* There is no fast optimal algorithm
* Ranking is expensive if similarity is computed for ALL documents 🡪 most docs have similarity score 0 for a query 🡪 potential to speed things up
* In practice: use heuristics to prune search space

## Results presentation

* Most often: list of links with doc description
* User can identify good hits based on description 🡪 no need to view the document
* Show:
  + Doc title
  + url
  + metadata
  + summary
    - static
      * always the same, independent of the query
      * first 50 words of document
      * Extract „key sentences“ (simple NLP heuristic)
      * Complex NLP to synthesize summary
    - Dynamic
      * Present snippets within the document that contain several of the query terms
      * Prefer snippets where query terms occur as a phrase
      * Summary contains all other words also around the query term hits
      * Cache documents 🡪 position in document available from positional index (cached copy can be outdated)