Foundations of Information Retrieval

Sebastian Hofstätter

sebastian.hofstaetter@tuwien.ac.at

TU WIEN

Today

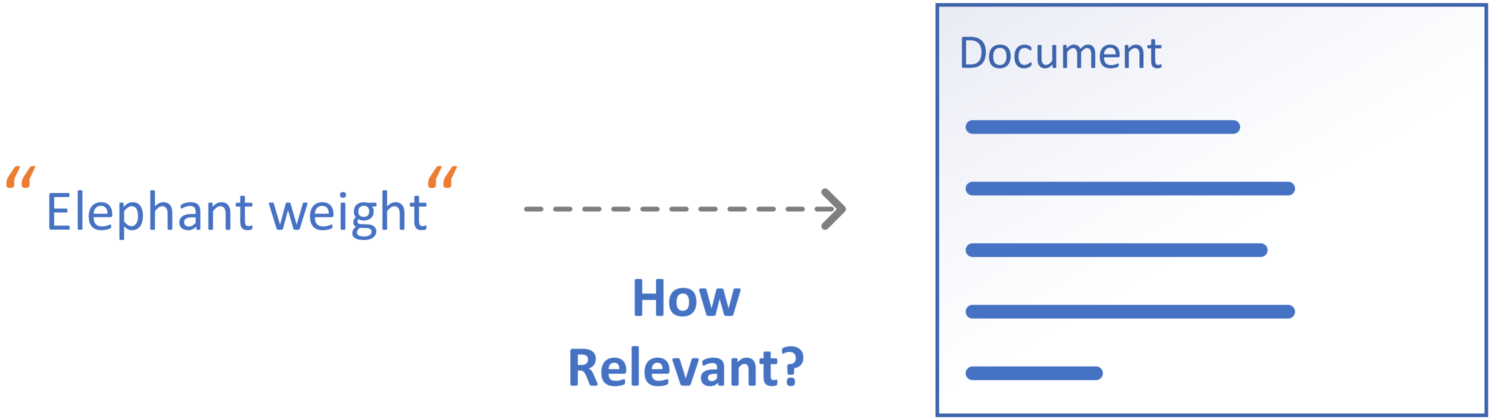
With a break after ~40 minutes

Foundations of Information Retrieval

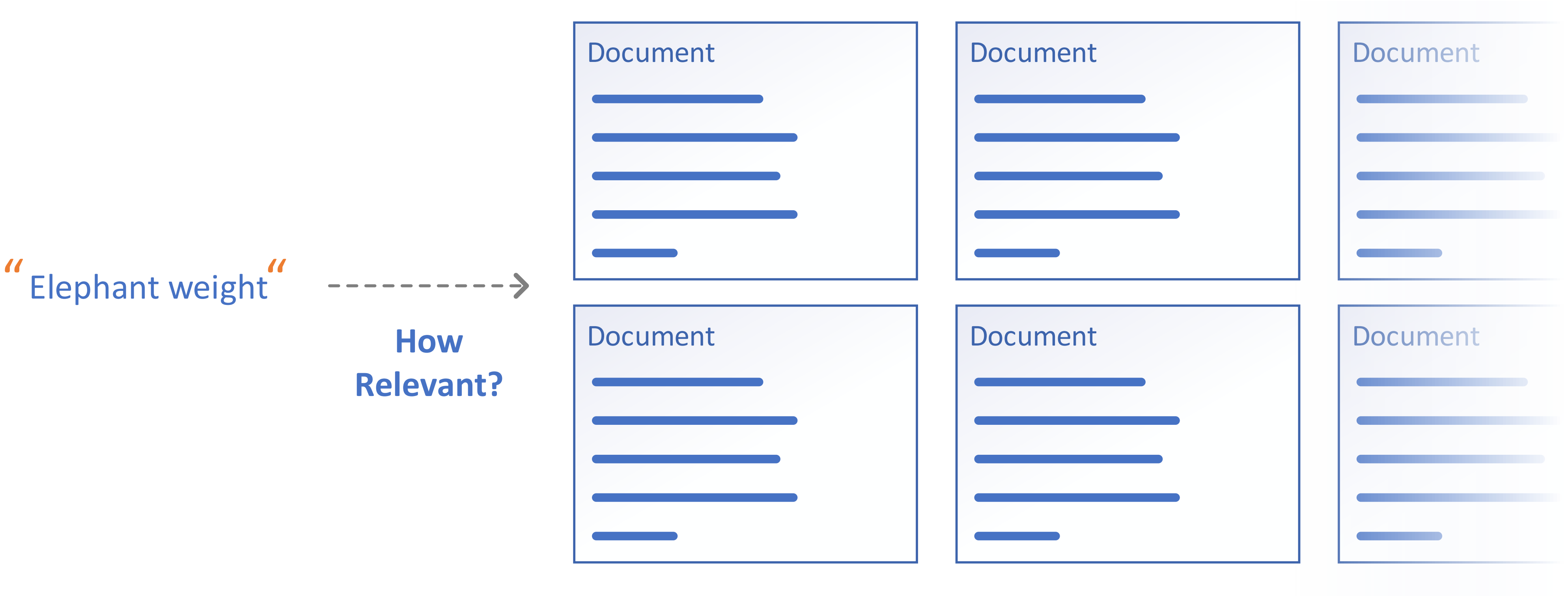
1. Inverted Index
2. Search
3. The Anatomy of a Large-Scale Hypertextual Web Search Engine

Some materials taken from: Introduction to IR by Manning lecture materials & Mihai Lupu's previous lecture slides

Information Retrieval



Information Retrieval (Finding the needle in the haystack)



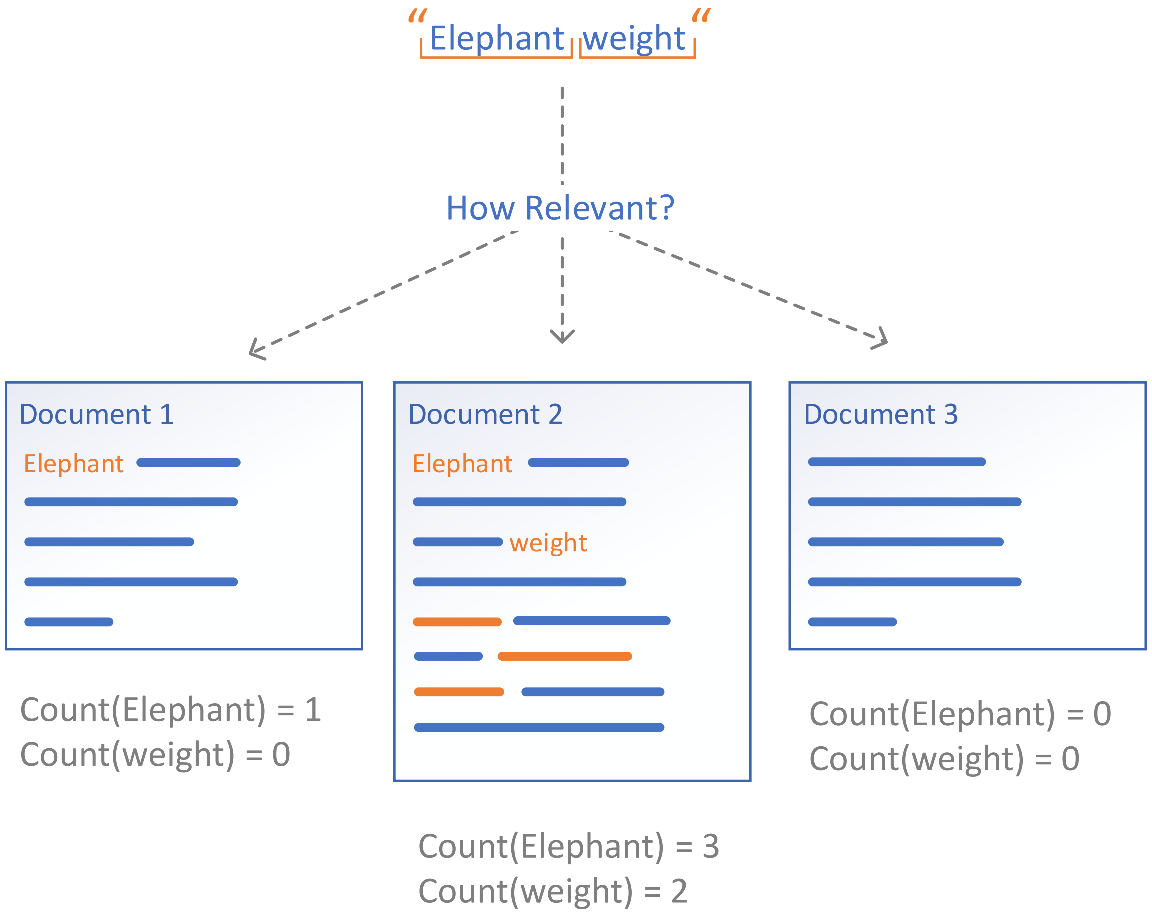
Notes on terminology

- **Documents** can be anything: a web page, word file, text file, article …
  (we assume it to be text for the moment)
  - A lot of details to look out for: encoding, language, hierarchy, fields, …

- **Collection**: A set of documents (we assume it to be static for the moment)

- **Relevance**: Does a document satisfy the information need of the user and does it help complete the user's task?

Relevance (based on text content)



- If a word appears more often -> more relevant
- Solution: count the words
- If a document is longer, words will tend to appear more often -> take into account the document length
- Counting only when we have a query is inefficient

Details of scoring models in the scoring lecture

Inverted Index

Transforming text-based information

Inverted Index

- Inverted index allows to efficiently retrieve documents from large collections
- Inverted index stores all statistics per term (that the scoring model needs)
  - **Document frequency:** how many documents contain the term
  - **Term frequency per document:** how often does the term appear per document
  - Document length
  - Average document length
- Save statistics in a format that is accessible **by a given term**
- Save metadata of a document (Name, location of the full text, etc..)

Inverted Index

Document data

```
Document Ids & Metadata:

[0] = (”Wildlife”, ”location”,...)
[1] = (”Zoo Vienna” ,...)
...

Document Lengths:

[0] = 231 [1] = 381 ...
```

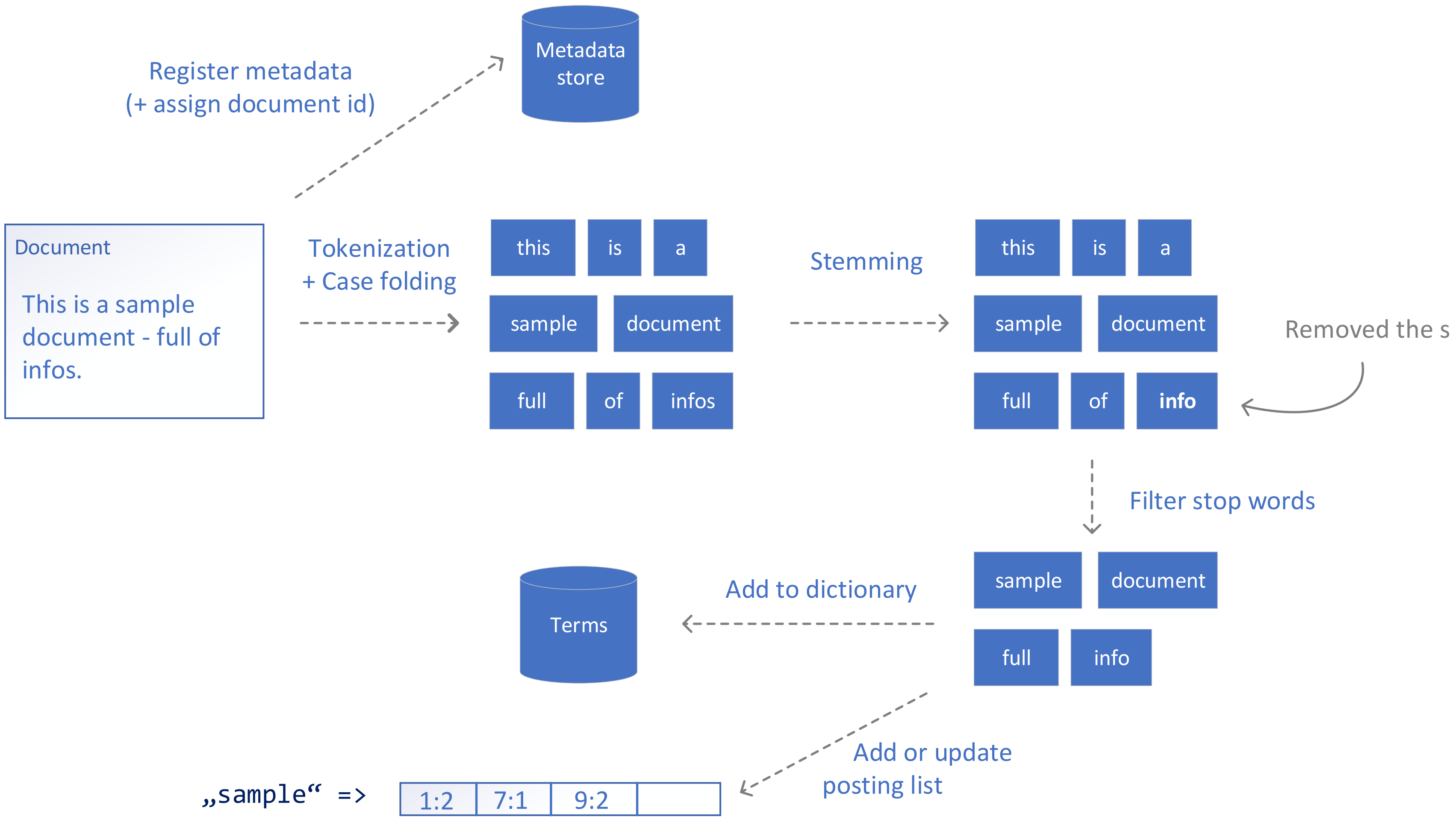
Term data

```
”elephant“ =>  | 1:5 | 2:1 | 3:5 | 4:5 | … |
”lion“ =>      | 1:2 | 7:1 | 9:2 | … |
”weight“ =>    | 4:1 | 6:4 | … |
...
```

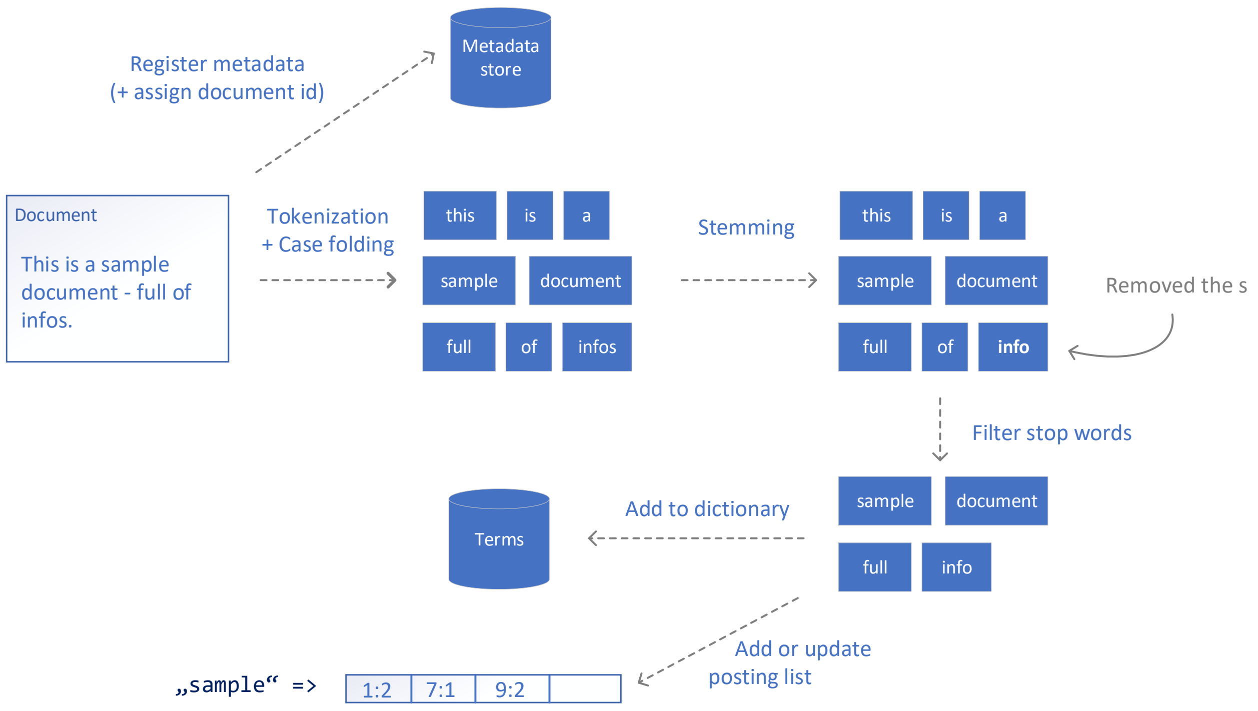
DocId　Term Frequency

- Every document gets an internal document id
- Term dictionary is saved as a search friendly data structure (more on that later)
- Term Frequencies are stored in a “posting list” = a list of doc id, frequency pairs

Creating the Inverted Index



Creating the Inverted Index



- Simplified example pipeline
- Linguistic models are language dependent
- A query text and a document text both have to undergo the same steps

Tokenization

- Transform a list of characters into a list of tokens
- A Token is itself an instance of a list of characters
- Each token is a candidate for an added term in the index
- How to split the stream of text into tokens?

Tokenization

- Naïve baseline: split on each whitespace and punctuation character
  - This splits U.S.A to [U,S,A] or 25.9.2018 to [25,9,2018]
  - Still a good baseline for English

- Improvement: keep abbreviations, names, numbers together as one token
  - Open source tools like Stanford tokenizer https://nlp.stanford.edu/software/tokenizer.shtml
  - Can also handle emoji 👌👍

Tokenization: Language issues

- French
  - ***L'ensemble*** $\rightarrow$ one token or two?
    - ***L*** ? ***L'*** ? ***Le*** ?
    - Want ***l'ensemble*** to match with ***un ensemble***
- German noun compounds are not segmented
  - ***Lebensversicherungsgesellschaftsangestellter***
  - 'life insurance company employee'
  - German retrieval systems benefit greatly from a **compound splitter** module
    - Can give a 15% performance boost for German

Tokenization: Language issues

- 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
- Further complicated in Japanese, with multiple alphabets intermingled
  - Dates/amounts in multiple formats

フォーチュン500社は情報不足のため時間あた$500K(約6,000万円)

Katakana

Hiragana

Kanji

End-user can express query entirely in hiragana!

Stemming

- Reduce terms to their "roots" before indexing
- "Stemming" suggests crude affix chopping
  - language dependent
  - ***automate(s), automatic, automation*** all reduced to ***automat***.

- More advanced form: **Lemmatization**: Reduce inflectional/variant forms to base form (*am, are, is* $\rightarrow$ *be*)
  - Computationally more expensive

Stemming: Porter’s algorithm

- Common algorithm for stemming English text

- 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.*

A lot of details at: [*http://snowball.tartarus.org/algorithms/porter/stemmer.html*](http://snowball.tartarus.org/algorithms/porter/stemmer.html)

Normalization

- Normalize words in the index

- Abbreviations: We want to match ***U.S.A.*** = ***USA***
- Accents: e.g., French ***résumé = resume*****.**
- Umlauts: e.g., German: ***Tuebingen*** = ***Tübingen***

- Can be very domain-specific

Case folding

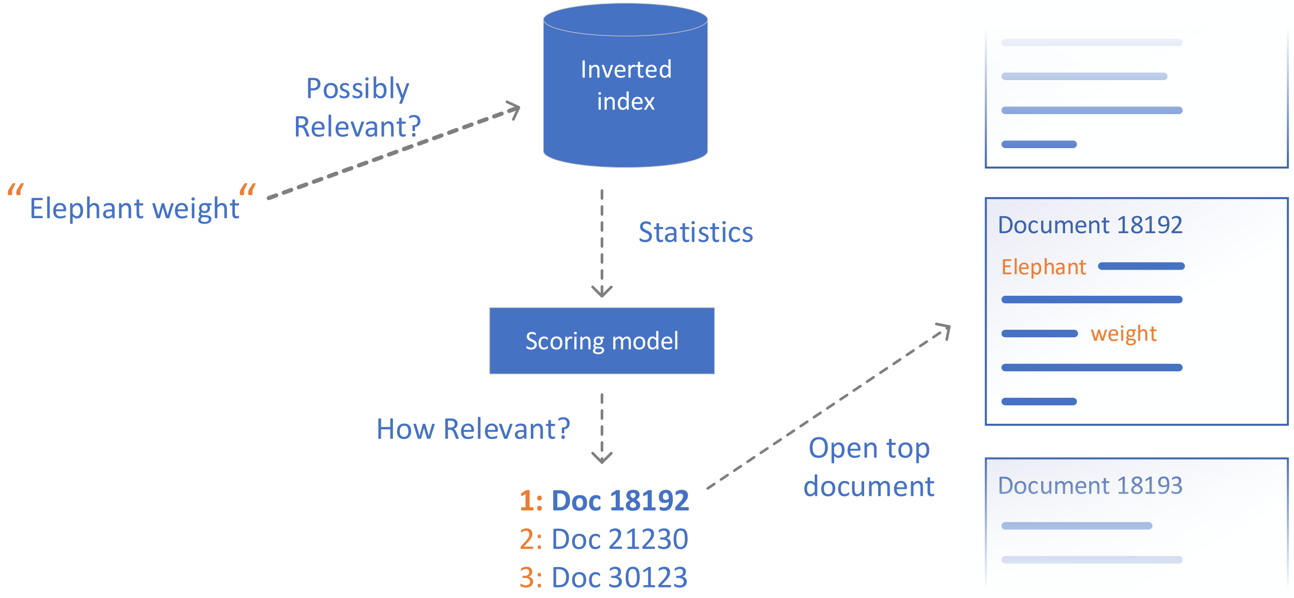
- Reduce all letters to lower case
  - Allows to match more occurrences

- This removes precise information about names, abbreviations etc…
  - A possible solution is to store two versions – one lowercased and one original The usefulness of this depends on the user entering a query in the correct casing

Search

Efficiently searching with the Inverted Index

Querying the Inverted Index



- No need to read full documents
- Only operate on frequency numbers of potentially relevant documents*
- Sort documents based on relevance score – retrieve most relevant documents

* it's not that easy because a document could be relevant without containing the exact query terms – but for now keep it simple

Types of queries (including, but not limited to)

- **Exact matching**: match full words and concatenate multiple query words with “or”
- **Boolean queries**: “and” / “or” / “not” operators between words
- **Expanded queries**: automatically incorporate synonyms and other similar or relevant words into the query
- **Wildcard queries, phrase queries, phonetic queries** (e.g. Soundex) …

Boolean queries

- Ask a query with Boolean operators: **and** / **or** / **not**

  `A and B ; (A and B) or C ; A and (not B) ...`

- Lucene: allows to plug in any other query type in A,B,C
  - And there are a lot of built in query types to choose from: https://lucene.apache.org/core/7_0_0/core/org/apache/lucene/search/Query.html

Related in name:

- Boolean Retrieval Model:
  - Simple form of retrieval without relevance ranking
  - Just binary information if the word is or is not in a document

(more in the IR book by Manning: https://nlp.stanford.edu/IR-book/html/htmledition/processing-boolean-queries-1.html)

Wildcard queries

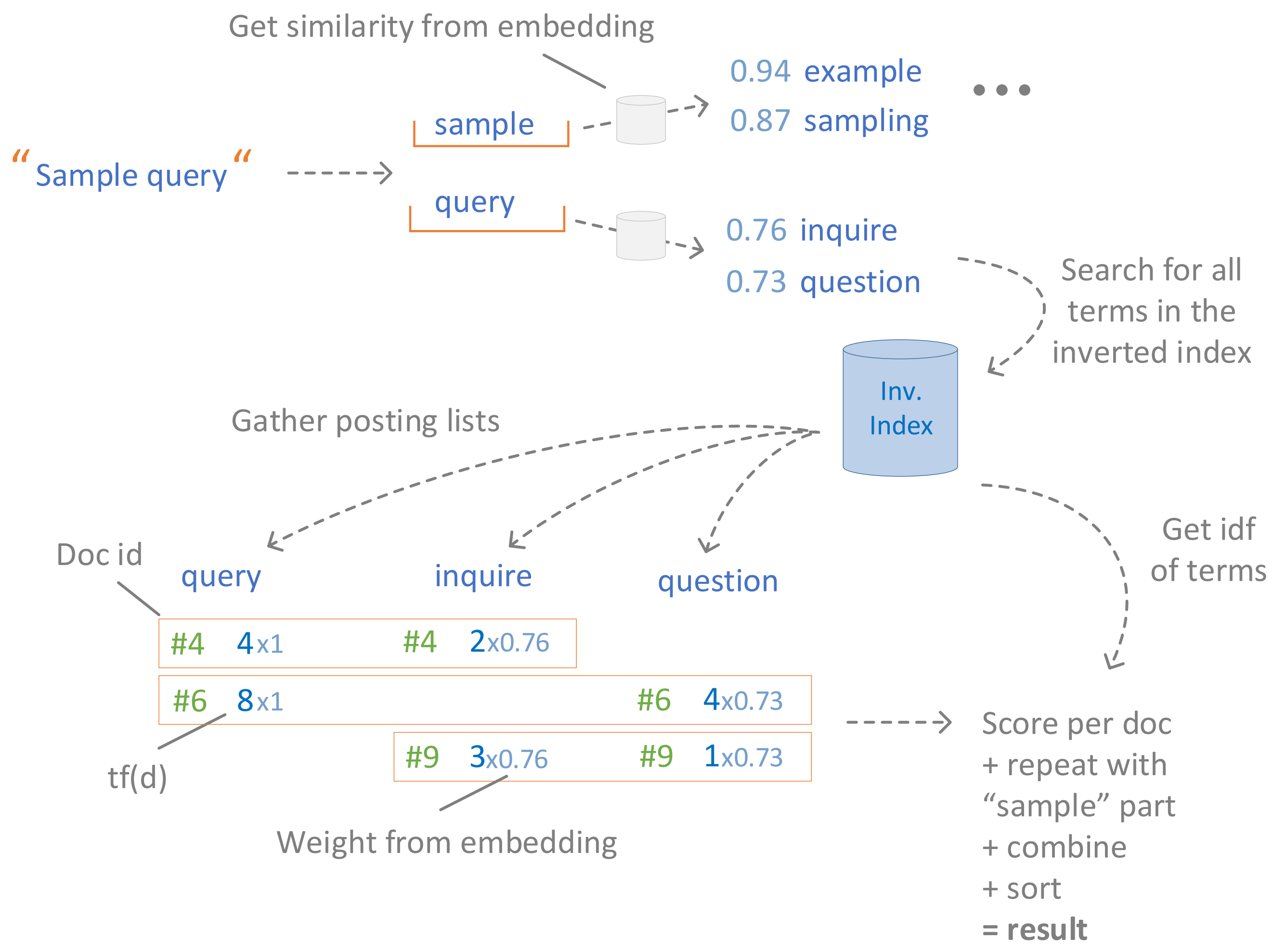
- Only specify part of a word you want to search for:
- Simple autocomplete: comp* -> computer, computation, compiler …
  - * stands for any possible characters the index knows about
  - Good with a tree-like dictionary, where we follow all branches after the known characters
- Can become computationally very expensive
  - Especially if the * is at the beginning & in the middle
  - Mitigated by specialized index architectures like:
    Permuterm and k-gram indexes (more in the IR book by Manning)

Query expansion

- Search including additional words not part of the query
- Added words need to be topically related, not only synonyms
- This allows to retrieve/boost relevant documents without the actual query present in the document

- Data could be from a variety of sources:
  - Handcrafted synonyms, abbreviations: e.g. WordNet
  - Learned from previous search user sessions: Only possible for big user bases
  - Unsupervised learned from a word embedding: Encoding relationships between words in vectors and taking the nearest neighbors

Query expansion: Research Example



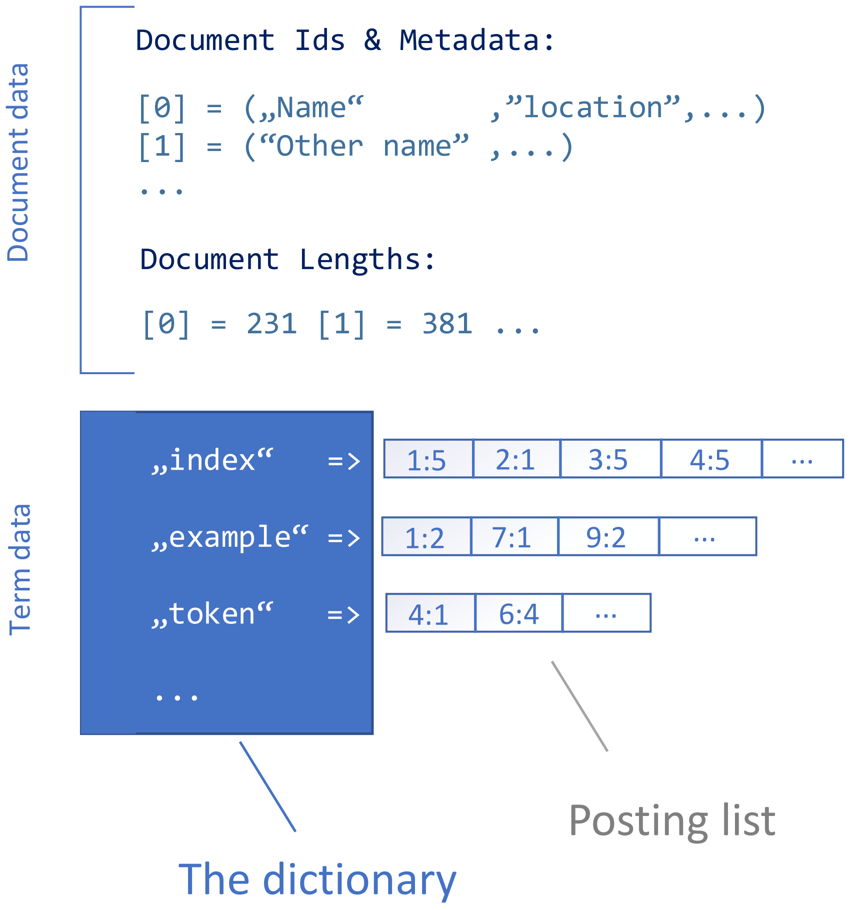
- Using a Word Embedding to automatically expand with similar words
- Adapted relevance model to score 1 document with multiple similar words together

N. Rekabsaz, M. Lupu, A. Hanbury, and G. Zuccon, "Generalizing Translation Models in the Probabilistic Relevance Framework," CIKM 2016
https://dl.acm.org/citation.cfm?id=2983833

Break

5 minutes or so

Inverted Index: Dictionary



- Dictionary<T> maps text to T
  - T is a posting list or potentially other data about the term depending on the index
- Wanted properties:
  - Random lookup
  - Fast (creation & especially lookup)
  - Memory efficient (keep the complete dictionary in memory)
- Naturally, there are a lot of choices

Dictionary data structures

- **Hash table**: Maps the hash value of a word to a position in a table

- **Trie** (or Prefix Tree): stores alphabet per node and path forms word
- **B-Tree:** Self balancing tree, can have more than two child nodes

- **Finite State Transducer** (FST): Memory friendly automaton

Related:

- **Bloom Filter**: Test if an element is in a set (false positives possible)

Hash table

- Uses a hash function to quickly map a key to a value
  - Collisions possible, have to be dealt with (quite a few options)
- Allows for fast lookup: O(1) (this doesn't mean it is free!)
- No sorting or sorted sequential access
- Does only a direct mapping
  - No wildcards – no autocomplete

Trie

- Tree structure with one character key per node
  and as many children as available characters per node
  (in it’s simplest form)
  - At the beginning every next character pointer is null
  - When a word is inserted in the Trie structure: for every character a new node is added recursively, each deeper level corresponds to the next char index
- A path in the Trie represents a word
- Not feasible for large character sets (No emoji support 😢)
  - There are versions that mitigate this problem

For the curious: https://medium.com/basecs/trying-to-understand-tries-3ec6bede0014

B-Tree (an its variants: B+, B* ..)

- Self balancing tree with multiple children per node
- The same height for all leaves

- Logarithmic time access (add, lookup)
  - Can be implemented very cache friendly (one node contains multiple keys)
  - B+: Allows for fast sequential access (if leaves are connected with pointers)

- Also heavily used in relational database indexes, file systems

Fun fact: The “B” in B-Tree has no official meaning

B-Tree with Prefixes

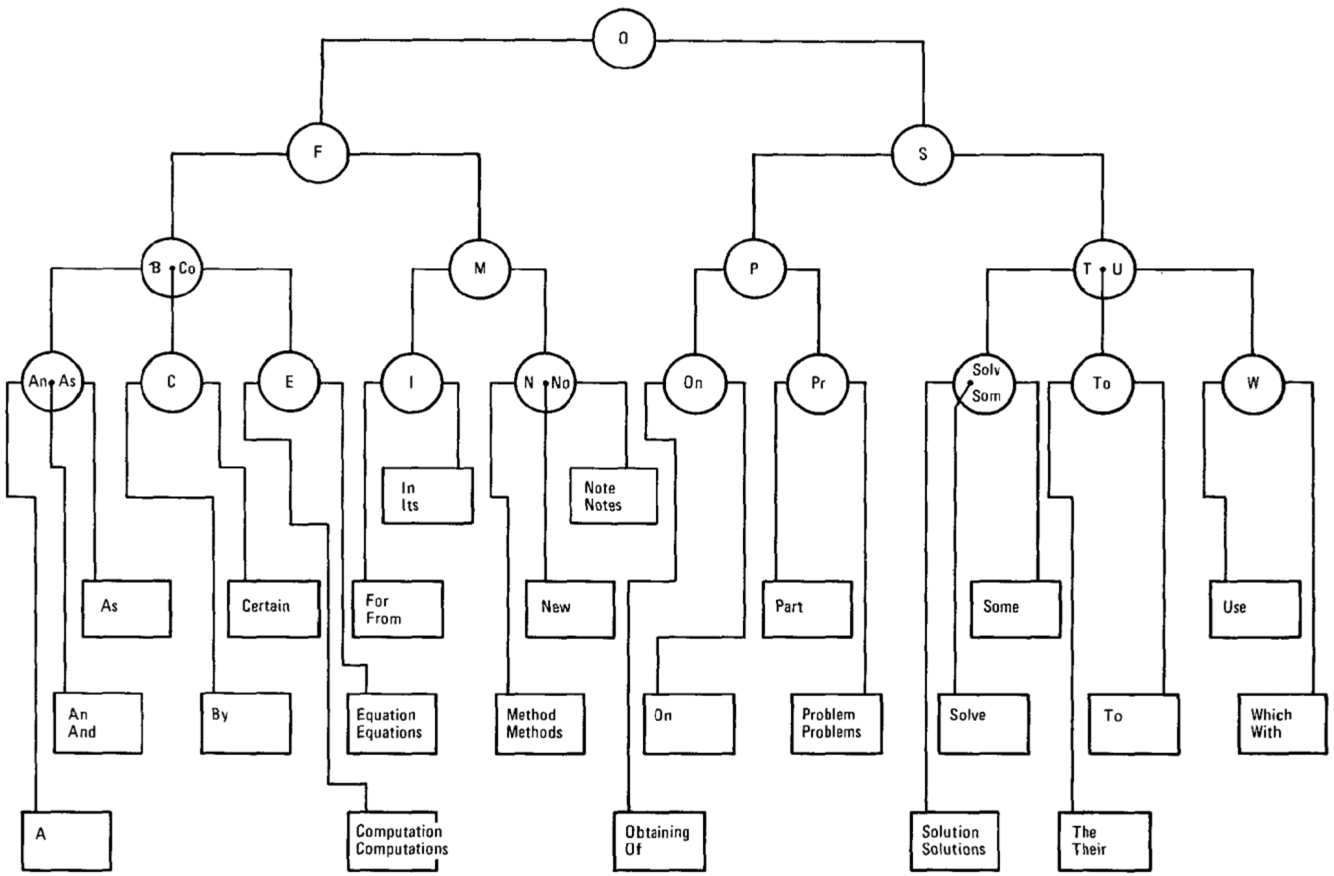


Fig. 1. Example of a simple prefix *B*-tree

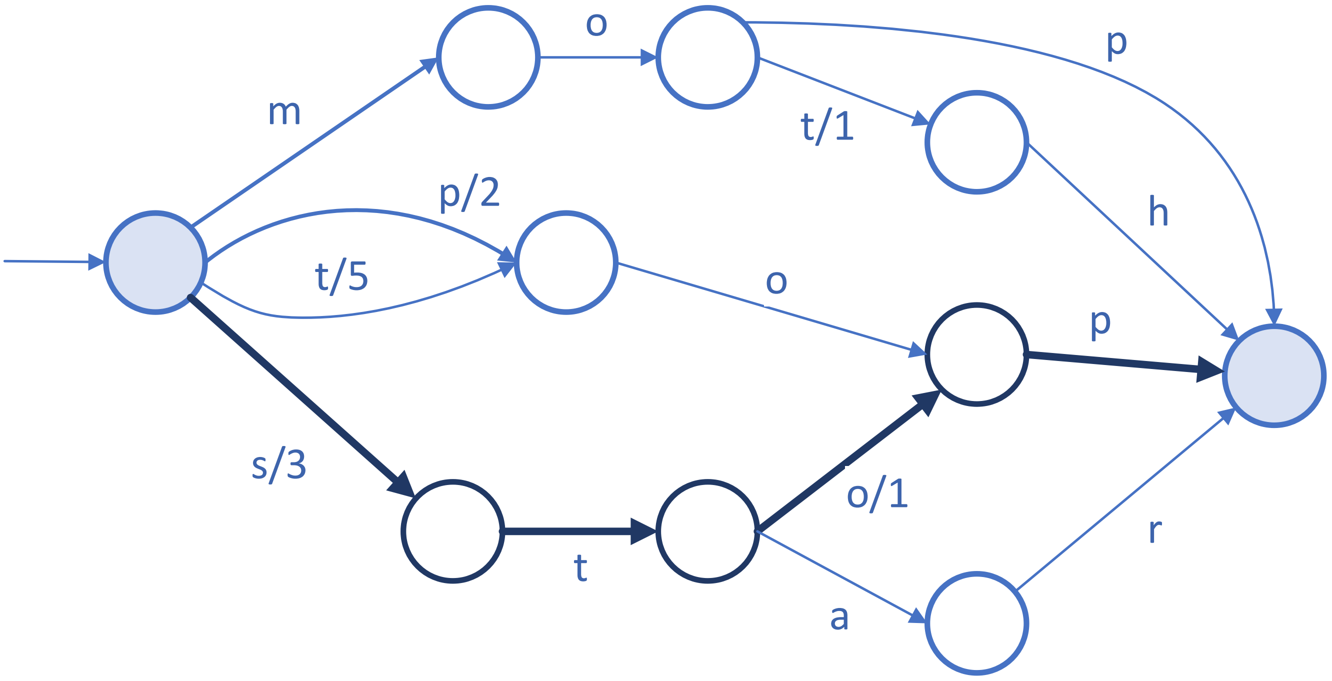
- At the upper nodes, we don't need to compare the whole string as key
- Split nodes with the shortest separator string
- Full strings are still sequentially + sorted accessible

Prefix B-Trees, Bayer and Unterauer TODS 1977
https://dl.acm.org/citation.cfm?id=320530

Finite State Transducer

- Automaton, where arcs encode characters and parts of the output
  - Output is summed up while traversing the arcs
- Very memory efficient dictionary index structure
  - 70mb for Wikipedia dictionary
- Used in Lucene as primary search dictionary
  - Not used for indexing, rather build from existing sorted vocabulary
  - Stores terms and an address of the term data in another location

Finite State Transducer in Lucene



- FST maps the words: *[mop, moth, pop, star, stop, top]* to their index (0, 1, 2, ...)
- As you traverse the arcs sum up the outputs:
- ***stop*** hits 3 on the **s** and 1 on the **o**, so its output is 4.
- Output can be arbitrarily assigned

Figure and example from: http://blog.mikemccandless.com/2010/12/using-finite-state-transducers-in.html

Bloom filter

- Checks whether an element is in a set, does not store the actual items
- Probabilistic data structure
  - False positives possible, but no false negatives
  - False positive probability depends on filter size, set size, hash functions
- Can be used to check if an expensive request to e.g. a data storage at another machine is going to yield a result

Definition:

- False positive: Predicts a yes, although the element is actually a no

Good article with further information: https://en.wikipedia.org/wiki/Precision_and_recall

Bloom filter

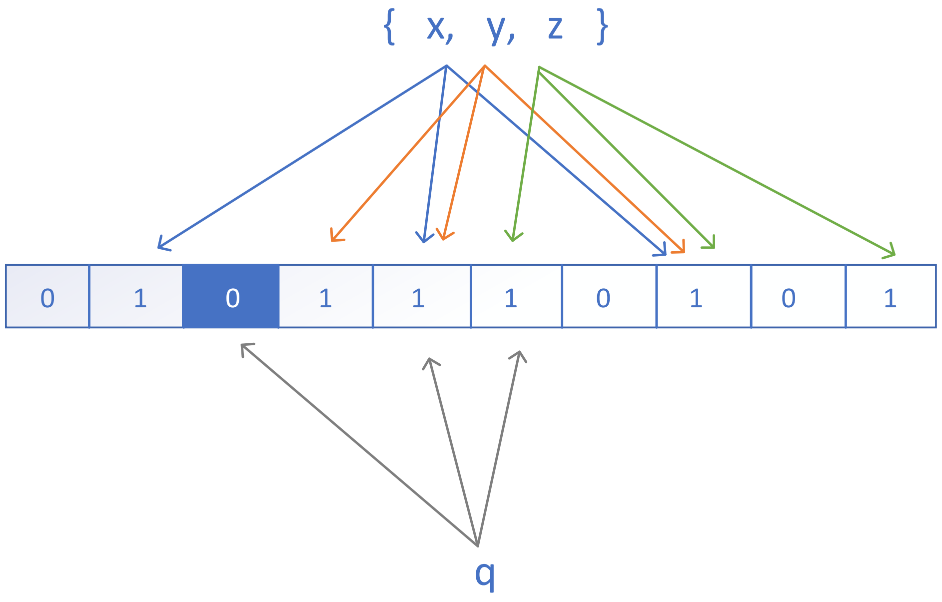


Figure from: https://en.wikipedia.org/wiki/Bloom_filter

- Data store: a bit array
- Bit array is all 0 at the beginning
- Different (here 3) hash functions, each returns a position in the array
  - Set bits to 1 for elements (x, y, z) in these positions
  - Positions can overlap
- Lookup (q): Check if positions returned by hash functions are all 1, if not element is definitely not in the set

Spell-checking

- Two principal uses
  - Correcting documents being indexed
  - Correcting user queries to retrieve correct answers – e.g. did you mean .. ?
- Two main flavors:
  - Isolated word
    - Check each word on its own for misspelling
    - Will not catch typos resulting in correctly spelled words
    - e.g., ***from*** $\rightarrow$ ***form***
  - Context-sensitive
    - Look at surrounding words,
    - e.g., ***I flew form Heathrow to Narita.***

Spell-checking by Peter Norvig

- Simple isolated spell-checking in a few lines of code
- Uses a text file of ~1 million words (from books)
  - For correct spelling information
  - Probability of each word occurring, if multiple correctly spelled candidates are available
- Get set of candidate words with: deletion or insertion of 1 char, swapping two adjacent chars, replace 1 char with 1 other
- Select most probable correct spelling from available candidates

Details (and implementation in various languages) here: https://norvig.com/spell-correct.html

The Anatomy of a Large-Scale Hypertextual Web Search Engine

In this paper, we present Google, a prototype of a large-scale search engine which makes heavy use of the structure present in hypertext.

1998: Google

- Started as a research project at Stanford
- Obviously a lot of good ideas
- Information retrieval as a problem of context and scale

Original paper: (highly recommended reading)

The Anatomy of a Large-Scale Hypertextual Web Search Engine, Brin and Page
http://ilpubs.stanford.edu:8090/361/

1998: Google – using context

- Using context inside the document (HTML tags + formatting)
- Using positional posting lists and proximity in the ranking
- Using links:
  - **PageRank**: Using the link graph between documents to assign a score to each document *(detailed explanation in the web search lecture)*
  - Use anchor (link) text as a description of the page it points to

Relevance beyond pure text matching

- PageRank, Localization, Speed … Google, Bing, etc. use 100s of values to rank results

- Values (Google calls them “Signals”) are combined to generate displayed ranking order
  - Learning to Rank: combine values with Machine Learning
  - Uses log data from previous users to learn better relevance scores
  - Active field of research to use recent advances in NLP (Natural Language Processing) to learn relationships between the query and the full text of the documents

Summary: Foundations of Information Retrieval

1. We save statistics about terms in an inverted index
2. The statistics in the index can be access by a given term (query)
3. The statistics are used to create a relevance score for a document

1. We save statistics about terms in an inverted index
2. The statistics in the index can be access by a given term (query)
3. The statistics are used to create a relevance score for a document

Thank You