

Multimedia Retrieval

Chapter 2: Text Retrieval

Dr. Roger Weber, roger.weber@ubs.com

- [2.1 Overview and Motivation](#)
- [2.2 Metadata](#)
- [2.3 Feature Extraction](#)
- [2.4 Text Retrieval Models](#)
- [2.5 Indexing Structures](#)
- [2.6 Lucene - Open Source Text Search](#)
- [2.7 Literature and Links](#)



2.1 Overview and Motivation

- Managing and retrieving information remains a challenging problem despite the impressive advances in computer science. The first generation of computers used punch cards to store and retrieve information, and memory and compute was precious. Many early algorithms hence have used Boolean models and brute-force approaches that quickly decide whether something is relevant or not. Today, memory and compute are extremely cheap, and we have more elaborated retrieval techniques to accelerate searches. Only recently, map-reduce and deep learning have gone back to the brute-force methods of the early days.
- Typical types of information retrieval:
 - **Database:** information is maintained in a structured way. **Queries refer to the structure of the data and define constraints** on the values (SQL as query language). Being structured, however, does not allow for quick retrieval across all data items with something like this:

```
SELECT * FROM * WHERE * like '%house%'
```
 - **Boolean Retrieval Systems:** Boolean models simplified matters: while scanning the data, we can decide whether an entry is relevant or not. There is no need to keep track and sort results later on. This was a huge advantage for early information systems (those with the punch cards and later with tapes) as they only had to filter out which data items were relevant based on a Boolean outcome. Even though rather simple, it is still a dominant retrieval model.
 - **Retrieval System with Ranking:** Basic Boolean retrieval suffers from the lack of a ranked list. A user is typically interested in a few, good answers but has not the time to go through all of the potential thousands of relevant documents. If you search a book in an online store, you expect the best matches to be at the top. Newer models, hence, try to **determine how relevant a document is for the user (in his given context) given the query.**

- **Vague Queries against Database:** this search type allows the user to specify soft constraints, i.e., vague query parts. For instance, if you want to buy a new computer, you may specify an “Intel Core i7” CPU, 32GB of memory, 1TB of SSD, and at least GTX-980 graphics card. And of course, you don’t want to pay more than \$1000. As you walk through the options, you may realize that you can’t satisfy all constraints and you compromise on some of them (e.g., replace SSD with HDD but now with 4TB). Vague queries are best executed with “fuzzy” retrieval models with a cost function that needs to be optimized (to satisfy the user’s demand as far as possible)
- **Natural Language Processing (NLP):** Consider a database with industrial parts for machines. A complex query may look as follows:
 - “Find bolts made of steel with a radius of 2.5 mm, a length of 10 cm implementing DIN 4711. The bolts should have a polished surface and can be used within an electronic engine.”

The challenge of the above query is that we are not actually looking for the keywords “radius”, “DIN”, or “polished”. Rather, the **keywords refer to constraints and to a context** expressed by the user. Recent improvements in Natural Language Processing (NLP) enabled systems to “decipher” such queries. Modern recommendation systems can chat with the user to obtain the context and then perform a search to answer the information need. We will, however, not look at such systems in this course, but lay a few foundations here and there.

- **Web Retrieval:** early (text) retrieval systems focused on searches over managed and controlled document collections. With the Web, search engines were **faced with spamming, bad quality, aggressive advertisements, fraud, malware, and click baits**. Many retrieval models failed completely in this uncontrolled environment. Web retrieval addresses many of these concerns and tries to find, among trillions of possible answers, the best few pages for your query. The sheer volume of information is a challenge in its own.

- **Multimedia Content:** with cheap storage and the digital transformation of enterprises and consumers, enormous amounts of multimedia data gets created every day (images, audio files, videos). The methods of text retrieval only work on the meta data but not on the signal information of the content. We still have a large **semantic gap** when searching for multimedia content, but recent improvements in deep learning techniques rapidly closed that gap. These techniques automatically label multimedia content to allow for simpler text (or speech) search over multimedia content and thereby bridging the semantic gap between the signal information and the user's intent.
- **Heterogeneous, Distributed, Autonomous Information Sources:** meta search is a generic problem: the user does not want to repeat a query against all information sources, but rather search once against all systems. In more complex setups, each system **may hold the answer to a part of the query and only the combination of all parts yields the best results.** We will consider more complex searches in later chapters.

- The Retrieval Problem

Given

- N text documents $\mathbb{D} = (D_1, \dots, D_N)$ and the Query Q of the user

Problem

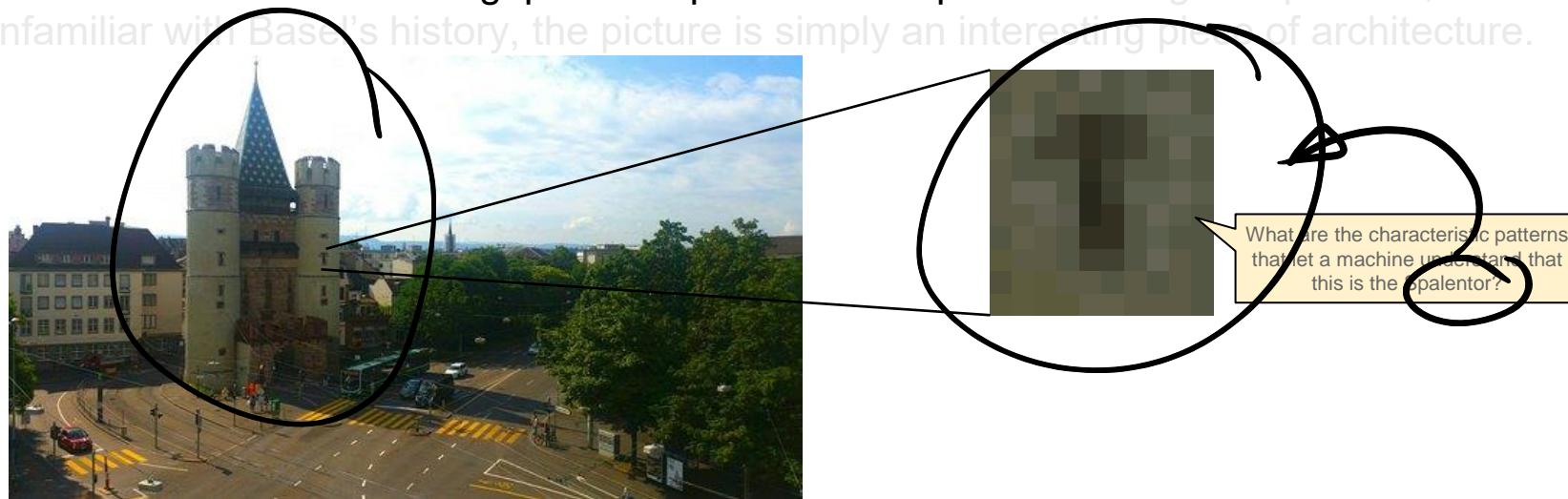
- find ranked list of documents which match the query well; ranking with respect to relevance of document to the query

- We will consider the following parts of the problem in this chapter:
 - Meta data extraction (annotations, manual, categories)
 - Feature extraction (simple methods only, next chapter looks into advanced methods)
 - Retrieval model (Boolean retrieval, vector space retrieval, probabilistic retrieval)
 - Index structures (inverted list, relational database)
 - Ranking of retrieved documents (RSV)
- We also look at a concrete implementation. **Lucene** is an open source project that provides reach text retrieval for many languages and environments.

2.2 Metadata

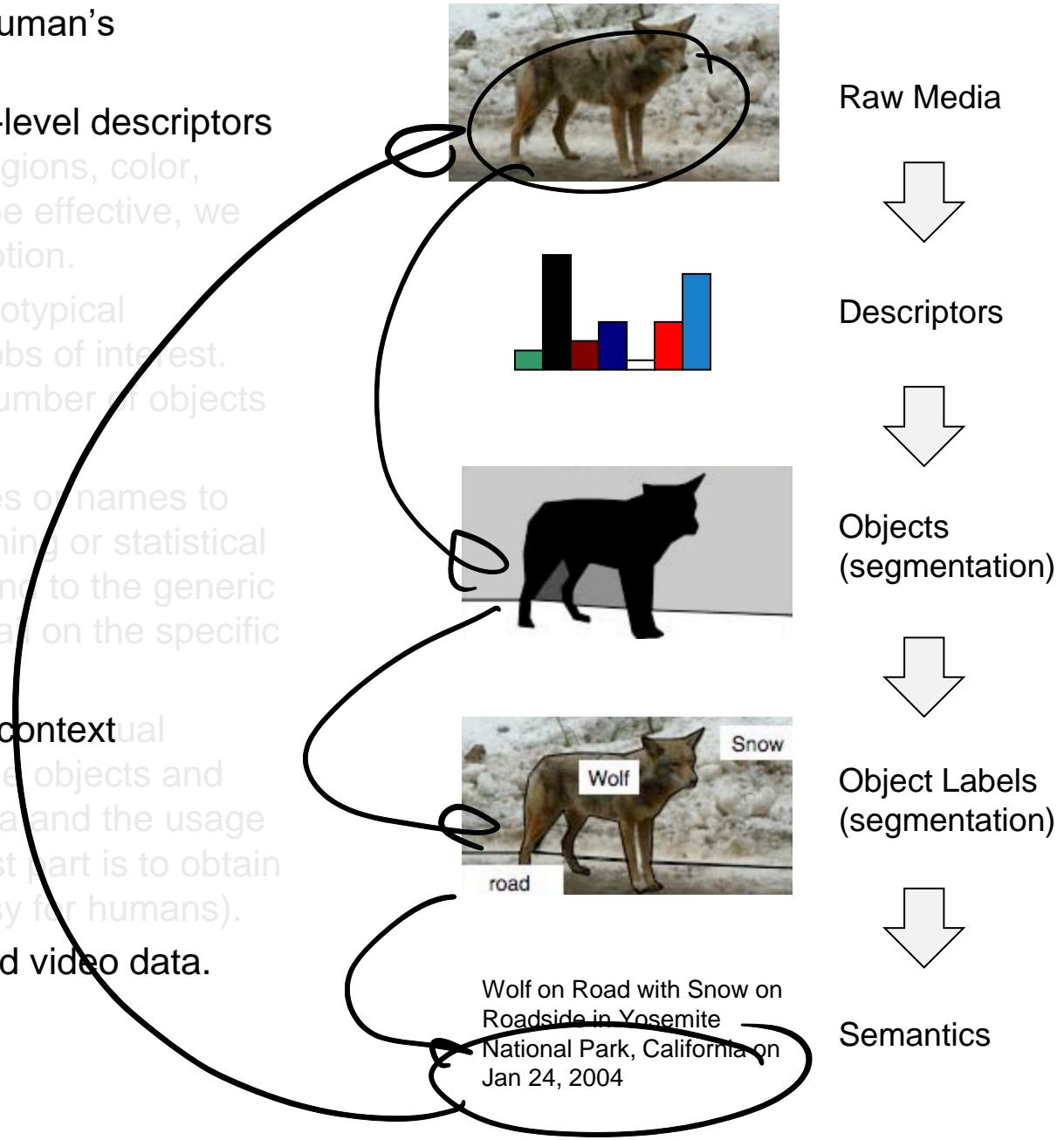
- With text and web retrieval, the descriptors for documents are the same as for user queries (words, phrases). Search performance is generally good even though we are just considering term occurrences. With other media types, it is not that simple. A user may want to query with natural language, but the documents do not contain keywords rather low-level signal information. This is known as the **Semantic Gap**.

- Consider the image below. For a machine, it contains pixels each with a color code attached to it. In some cases, additional meta-information may exist. For a person, it depicts the Spalentor in Basel. When looking for the Spalentor in images, we need to translate the term “Spalentor” somehow to the low-level signal information (or vice-versa). But which patterns in the picture let a machine understand that this is a picture relevant for the query “Spalentor”.
 - The semantic gap is the difference between the information extractable in an automated fashion from the raw image data and the interpretation of that same data by a person.
 - Also note that the semantic gap also depends on the person asking the question; for someone unfamiliar with Basel’s history, the picture is simply an interesting piece of architecture.

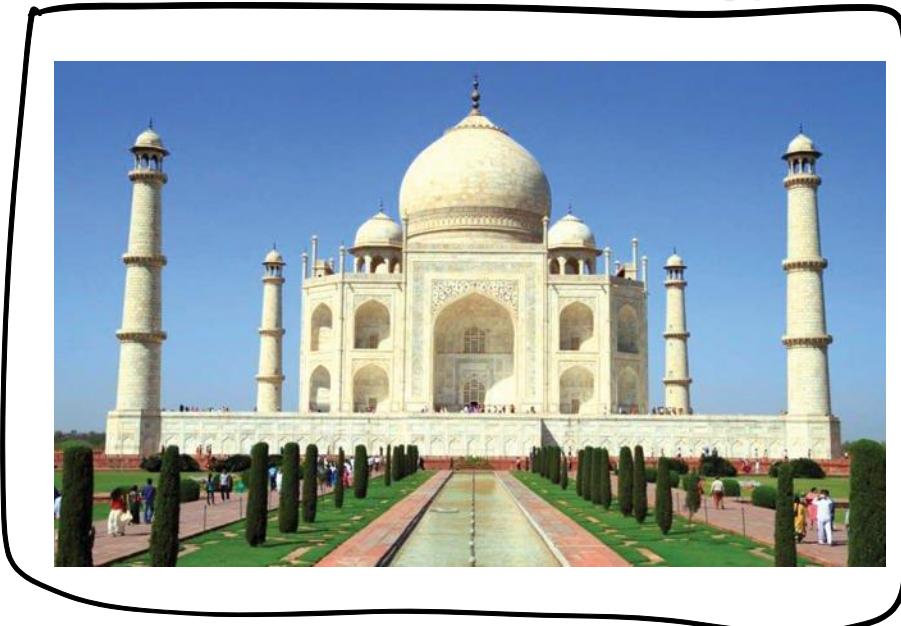


- **The same gap applies to audio files.** A user is not expressing a query at the signal level (amplitude, frequencies, etc.) but at a semantic level: “find me a rock ballad” or “funny comedian”.
- Humans interpret signal information in several steps:
 1. **Perception** – we are not measuring the physical quantities but rather obtain a “biased” perception that helps us to further infer information.
 - The eye is responding to three color channels and luminance. The **concept of color** is merely an **interpretation of our brain**, but it is essential to the next steps. Both eyes combined provide a spatial perspective of the scenery.
 - The ear is responding to **wave lengths** and measures delays between the ears to infer direction of the sound. The pre-processed signal that reaches the brain is **no longer physical quantities**.
 2. **Generic Semantic Inference** – the brain interprets the **perception** and enriches it with **semantic information**. The first step is poorly generic and is focused on important aspects (person, animal, sky, faces). At this stage, information hiding prevents over-stimulation of reasoning.
 3. **Specific Semantic Inference** – with our **knowledge**, **experience**, **cultural conditioning**, and **beliefs**, we infer contextual semantics including named objects (Spalendor), events (Soccer match), and abstract concepts (emotions, spatial, time).
 - This step **depends on the individual experience and knowledge** of a person. You will infer different semantics for a picture of your mother than someone who does not know her.
- **To close the semantic gap, a machine must address each of the three levels.** Content-Based Retrieval systems started with the perceptual level. Recently, deep learning made huge progress on the generic semantics and on the specific semantics. In between, we have classical retrieval on metadata obtained either by manual or automated processes. Metadata is matching the semantics of users much better and is still the dominating search paradigm.

- A retrieval system must mimic the human's interpretation of the low-level signal
 - The raw media is mapped to low-level descriptors that summarize information on regions, color, texture, or points of interest. To be effective, we need to replicate human's perception.
 - Object recognition combines prototypical descriptors and infers regions/blobs of interest. Image segmentation yielding a number of objects but without any classification.
 - Object labeling associates classes or names to objects often using machine learning or statistical approaches. The labels correspond to the generic semantics of users but may still fail on the specific interpretation of users.
 - Semantics result from additional contextual information either derived from the objects and their relation or through meta-data and the usage of a knowledge base. The hardest part is to obtain the context (which is also not easy for humans).
- Again, the same applies to audio and video data.



- We distinguish between two feature types going forward
 - Low level features that are based on the raw signal information and describe perception rather than semantics. Most of the early Content-Based Retrieval System were focused on low-level features and search paradigms like **Query by Example, Query by Sketch, or Query by Humming** As a general idea, these systems extract features from both the query and media objects, and perform a comparison to find best matches (similarity search, nearest neighbor search). The **semantic gap** is only closed with regard to perception; higher level gaps remain open and can challenge the user during the search (like this picture but need an other color for the car, or: can't sing correct but the tune is somehow like this).
 - High level features address **generic, specific, and abstract** semantic meaning. We can distinguish between **object, spatial, temporal, and event/activity** information. Further information encompasses **related concepts/objects, abstract concepts, and context**. For instance, let us consider the following picture of the Taj Mahal:



Object Facet	Value
Generic Object Instance	building, water, sky
Generic Object Class	mausoleum, tomb, dome, minaret
Specific Named Object Class	UNESCO World Heritage Site (since 1983)
Specific Named Object Instance	Taj Mahal

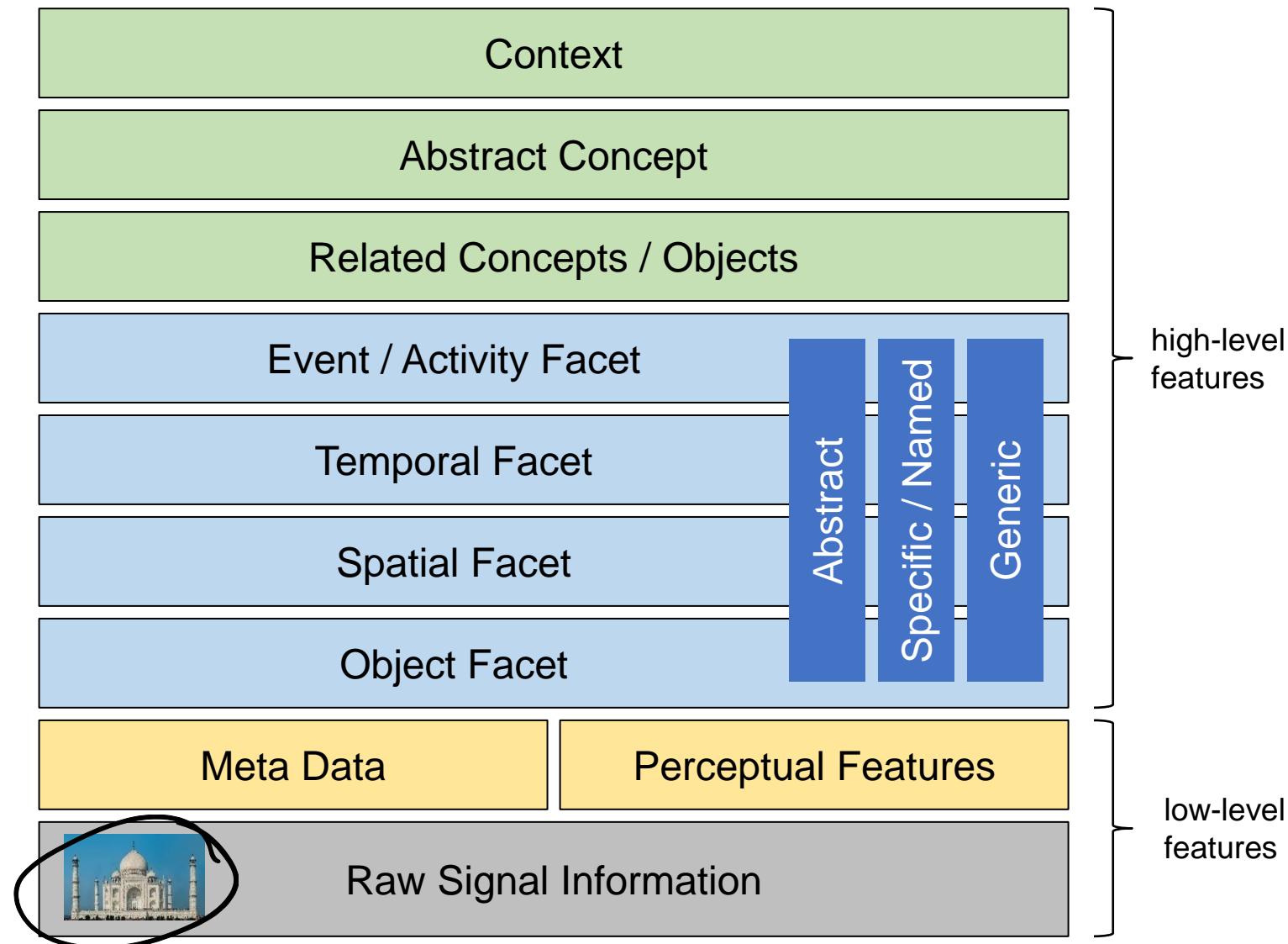
- Taj Mahal (contd)

Spatial Facet	Value
Generic Location	outside
Specific Location Hierarchy	India, Uttar Pradesh, Agra
Event / Activity Facet	Value
Generic Event/Activity	tourism, attraction
Specific Event Instance	International World Heritage Expert Meeting on Visual Integrity in 2006



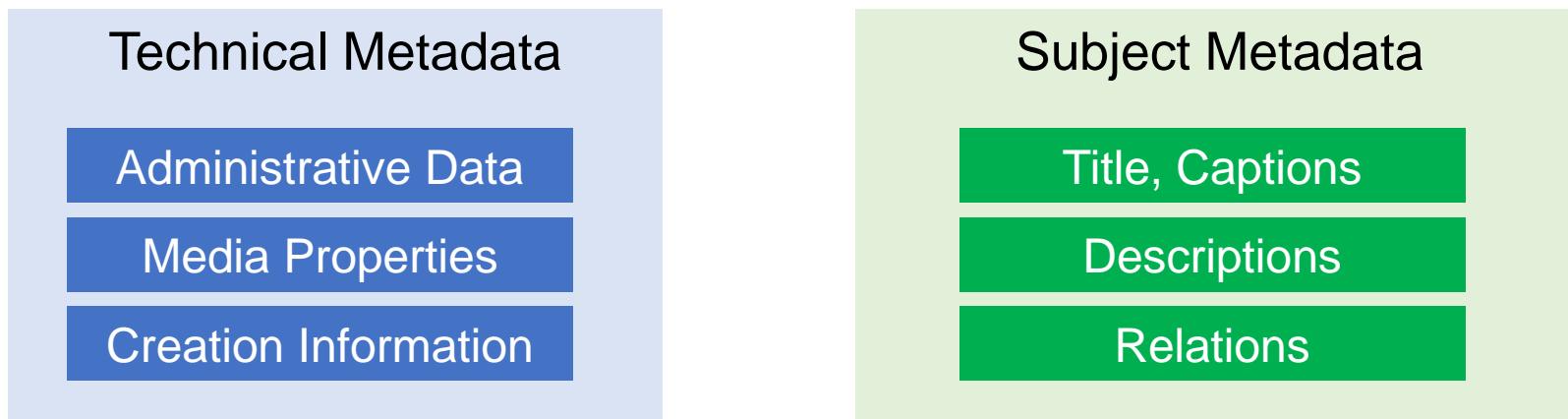
Temporal Facet	Value
Generic Time	summer, daytime
Specific Time	2006 (photo taken)
Contextual Facet	Value
Topic	Indian Architecture
Related Concepts / Objects	Shah Jehan, Mumtaz Mahal, Islam
Abstract Concept	love, death, devotion, remembrance
Context	built in memory of his favorite wife Mumtaz Mahal, by Shah Jehan; completed 1648

- In summary, to close the semantic gap we need to extract descriptors at different levels allowing a user to ask semantic queries. In this chapter, we start with the lower levels. The next chapter addresses some of the higher levels.



2.2.1 Metadata Extraction

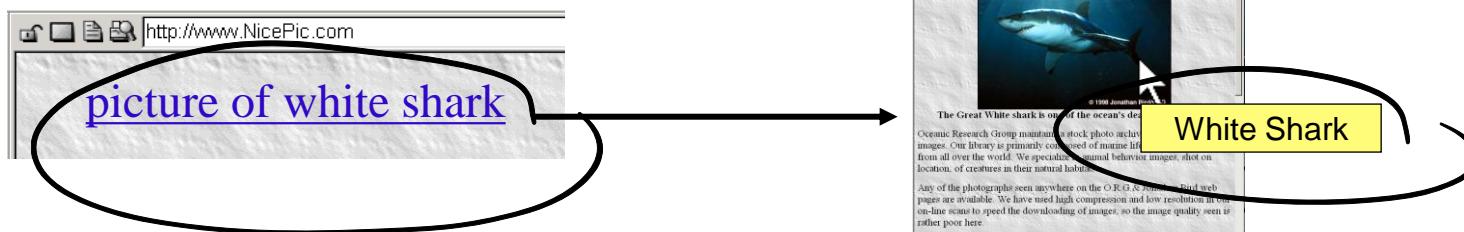
- There is a simple way to close the semantic gap: we annotate the media files with keywords and derive higher-level semantic features similar to the techniques we have seen in text and web retrieval. In this context, the **meta data** is a **low-level feature** in the form of structured or unstructured text, while the terms extracted and the reasoning on the **terms** denote the **higher level features** (which are not inferred directly from the raw signal).
- However, it costs about \$50 to \$100 to annotate an image with the necessary level of detail and quality. With the billions of images and the limited revenue generation from such annotations, this clearly is not an attractive path. Or would you pay \$100'000 for the 1'000 photos from your last vacation? Clearly not. So we need a cleverer approach to automate annotations as much as possible. This is not always feasible.
- We can divide meta data roughly into two groups:



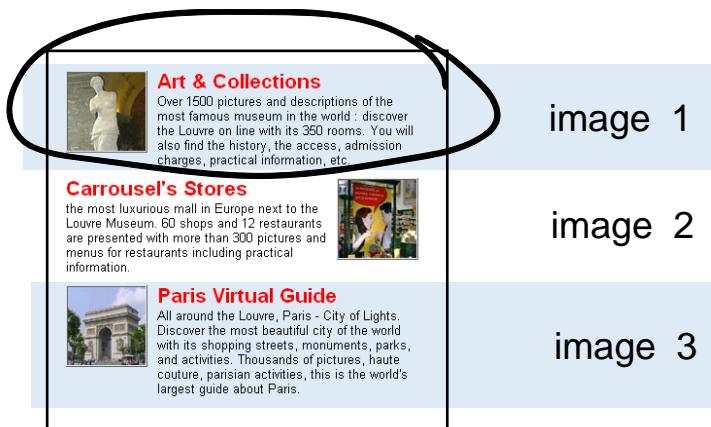
- There are many standards for metadata description like RDF, Dublin Core, Dublin Core Metadata Initiative and others that define standards how to annotate media files. They all are part of the semantic web initiatives to provide better connection of information. In the context of web pages, the meta-tag in the header holds all meta information about the current web page. Its format is: <meta name="description" content="text">. Next to description, a number of further meta data items are possible:

name	content
description	short description of web page
keywords	keywords associate with page
abstract	short narrative of content
author	author of this page
contact	contact person for this page
copyright	name of owner
dc.language	language of page (e.g., using RFC1766 and ISO 639)
dc.source	reference to page from which this page is derived
dc.creator	creator information for page
...12 more Dublin core tags and even more DCMI tags possible	

- In the context of multimedia content the web offers more information than the simple meta information in the header section. Similar to what we have seen in web retrieval, **links** and **embedding** in pages offer **further sources for meta data**
 - Link information (example: `img-tag` and `a-tag`)



- The `alt-attribute` in the `img-tag` is a good source for a **caption**. Sometimes the file name yields additional keywords of interest
- Hypertexts annotate the **referenced image** (like we did for web pages) with additional keywords. These annotations contain **keywords at different semantic levels**. If an image is frequently referenced, we may find a complete description of the content from various perspectives and covering a wide range of user specific semantics.
- A good source for keywords is the **surrounding area** on the web page. If we look before and after the image we find title, caption, and relevant keywords for the image. The same applies to links (also within the same page) to media objects. The surrounding area holds many interesting aspects.
 - What means surrounding? and how far does it stretch? This may also lead to false annotations



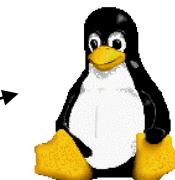
- Extracting information from the web page (basics)
 - The meta information of the web page is a good source for descriptors of an embedded image. In addition, headings or table headers before the image may contain further relevant information. The larger the document, the less likely such association may hold true
 - The window (in terms of characters in the HTML file) around the embedding holds many text pieces of potential relevance for the image. The size of the window must be carefully chosen to avoid wrong associations. Alternatively, we can weigh terms inversely to their distance to the embedding tag.

```

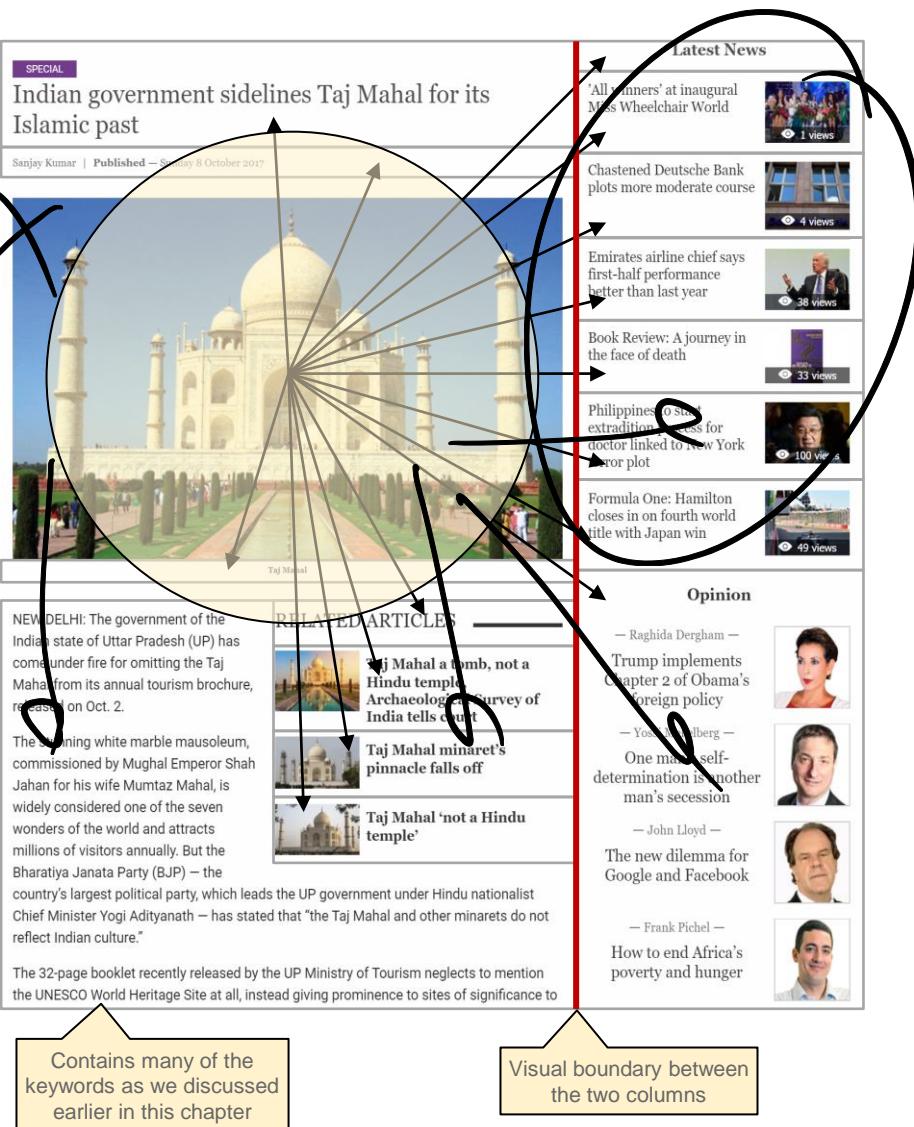
<HTML><HEAD>
  <TITLE>Linux is cool.</TITLE>
</HEAD>
<BODY BACKGROUND="../images/paper11.jpg">
<CENTER><H1>LINUX</H1>
<P>
  <IMG SRC="../images/tux.gif" ALT="picture the penguin from linux">
  <EM>This penguin, Tux, is the
      official mascot of Linux.</EM></CENTER>

<H2>MY &nbsp;&nbsp;FEELINGS&nbsp;&nbsp;ABOUT&nbsp;&nbsp;LINUX</H2>
I'll tell you, Linux has to be, ...
<P>
<H2>MY INVOLVEMENT&nbsp;&nbsp;WITH&nbsp;&nbsp;LINUX</H2>
...
</BODY>/HTML>

```



Source	Text
src-attribute	tux.gif
alt-attribute	picture the penguin from linux
title	Linux is cool.
h1	LINUX
em	This penguin, Tux, is the official mascot of Linux.
text	LINUX This penguin, Tux, is the official mascot of Linux. MY FEELINGS ABOUT LINUX



- An alternative approach uses **visual closeness** to annotate objects:
 - Instead of defining the neighborhood in the source code, it is defined by the **proximity in the visual layout of the page** (distance as perceived by reader)
 - Implementation:
 - Render the page and define core blocks on the page given the core structural elements (div, p, table, form, ...)
 - Compute distances between these blocks and the embedded object. The distance can be any measure like pt or pixel.
 - Add penalties to the distance if there is a (visual) delimiter between the blocks. For instance, a line separating table cells. Column boundaries in a multi-column layout. Other blocks in between.
 - Define a neighborhood and add all blocks intersecting with that neighborhood. Use the distance as a weight for the terms found within a block. Apply further weighting based on visual attributes such as bold, italic, header, ...
 - Summarize descriptions with bag-of-words approach and associate it to the image.

- A more targeted approach is to “scrape” information on media objects, especially if they are **highly standardized and categorized**. With images, this is hardly achievable and only for sets of official catalogues. But for **music and videos**, this is the most common approach. Consider you want additional annotations for your music library to be able to find songs by keywords. A good starting point is **MusicBrainz.org** which catalogues a large portion of published songs and is entirely public domain (you can download the entire database).
 - **Example** below: for every song in a media library, we can extract information about the artist, about albums and releases, and about individual songs and interpretations of it. Using services like **LyricWiki**, we can obtain a full description of high-level semantics for our songs. If you combine several services, you can easily complete the descriptions of your media library.
 - Both **IMDb** and **TMDb** offer similar services for movies and series. TMDb is a community built database and free to use (with usage restrictions as per license agreement)

Parapazzi
~ Release group by Lady Gaga

[Overview](#) [Aliases](#) [Tags](#) [Details](#) [Edit](#)

Wikipedia

"Parapazzi" is a song by American singer-songwriter Lady Gaga from her debut studio album, *The Fame* (2008). It was released as the fifth and final single by Interscope Records. Gaga wrote and produced the song with Rob Fusari. The song portrays Gaga's struggles in her quest for fame, as well as balancing success and love. Musically, it is an up-tempo techno-pop and dance-pop song whose lyrics describe a stalker following somebody to grab attention and fame.

Show more...
Continue reading at [Wikipedia...](#) Wikipedia content provided under the terms of the Creative Commons BY-SA license

Single

Release	Format	Tracks	Date	Country	Label	Catalog#	Barcode
Official							
Paparazzi	Digital Media	2	2009-07-02	GB			
Paparazzi	Digital Media	5	2009-07-05	GB			
Paparazzi	CD	2	2009-07-06	GB		602527121178	
Paparazzi: The Remixes	Digital Media	4	2009-09-08	US			
Paparazzi: The Remixes, Part Deux	Digital Media	3	2009-09-29	US	Interscope Records	0602527224169	
Paparazzi: The Remixes, Part Deux	CD	7	2009-10-13	US		602527217901	
Promotion							



Release group information
Artist: Lady Gaga
Type: Single

Rating
★★★★★ (see all ratings)

Tags
remix

External links
Discogs
Q1025916

Editing
Log in to edit

Last updated on 2014-01-07 09:00 UTC



We are the crowd
We're c-comin' out
Got my flash on, it's true
Need that picture of you
It's so magical
We'd be so fantastico

Leather and jeans
Gorge glamorous
Not sure what it means

But this photo of us, it don't have a price
Ready for those flashing lights
'Cause you know that baby, I

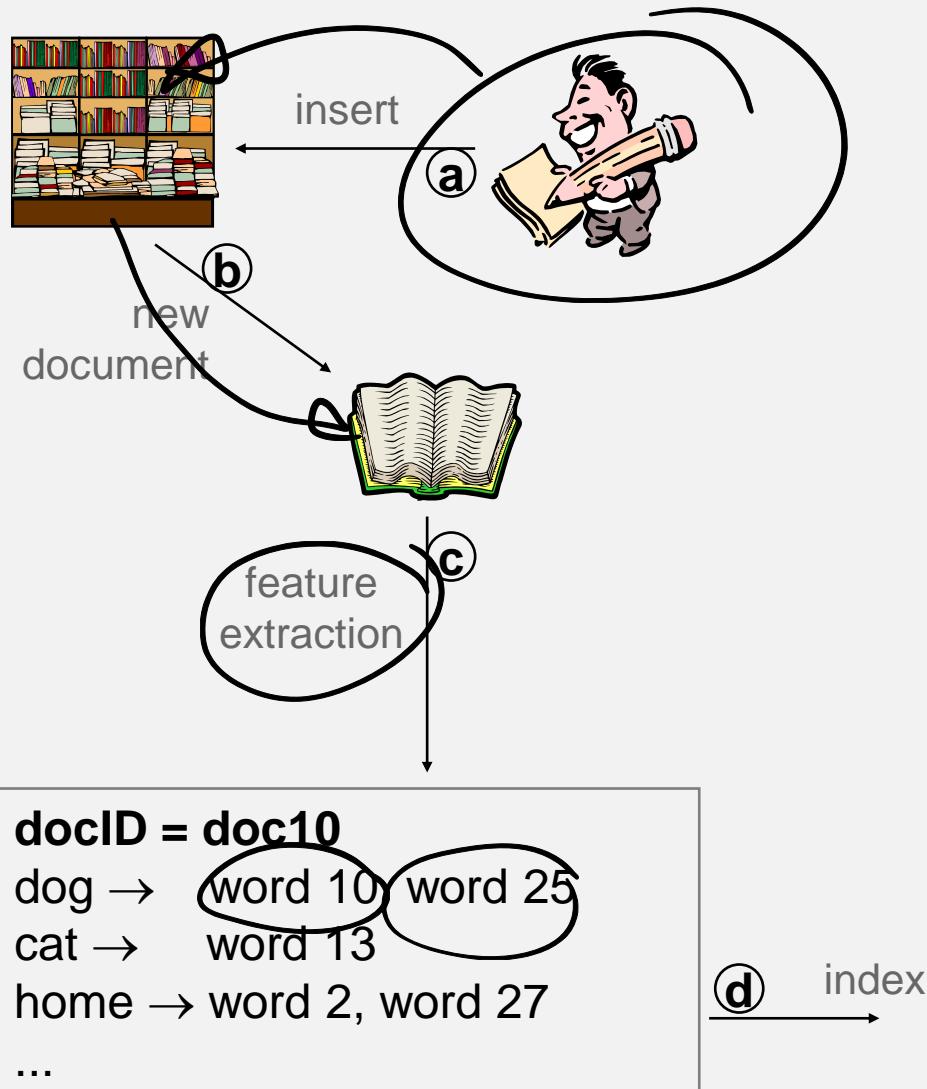
I'm your biggest fan
I'll follow you until you love me
Papa-paparazzi (ya-ha)
Baby, there's no other superstar
You know that I'll be
Your papa-paparazzi (ya-ha)

- Discussion: a good summary of the challenges around meta data is given by **Cory Doctorow** which he calls the **seven insurmountable obstacles** between the world as we know it and meta-utopia:
 - **People lie**. Metadata cannot be trusted because there are many unscrupulous content creators who publish misleading or dishonest metadata in order to draw traffic to their sites.
 - **People are lazy**. Most content publishers are not sufficiently motivated to carefully annotate all the content that they publish.
 - **People are stupid**. Most content publishers are not intelligent enough to effectively catalog the content they produce.
 - **Mission impossible—know thyself**. Metadata on the web cannot be trusted because there are many content creators who inadvertently publish misleading metadata.
 - **Schemas aren't neutral**. Classification schemes are subjective.
 - **Metrics influence results**. Competing metadata standards bodies will never agree.
 - **There's more than one way to describe something**. Resource description is subjective.

- Do we ignore meta data, then? Of course not, but we need to be careful what we are doing with the information provided. After all, a lot of the meta data can be extremely useful if the quality is right (see for instance MusicBrainz.org).
 - **Observational meta data** (automatically generated while crawling the web) is useful if it is hard to game the system (see PageRank as a good example).
 - **Need to take the trustworthiness** of the data provider into account

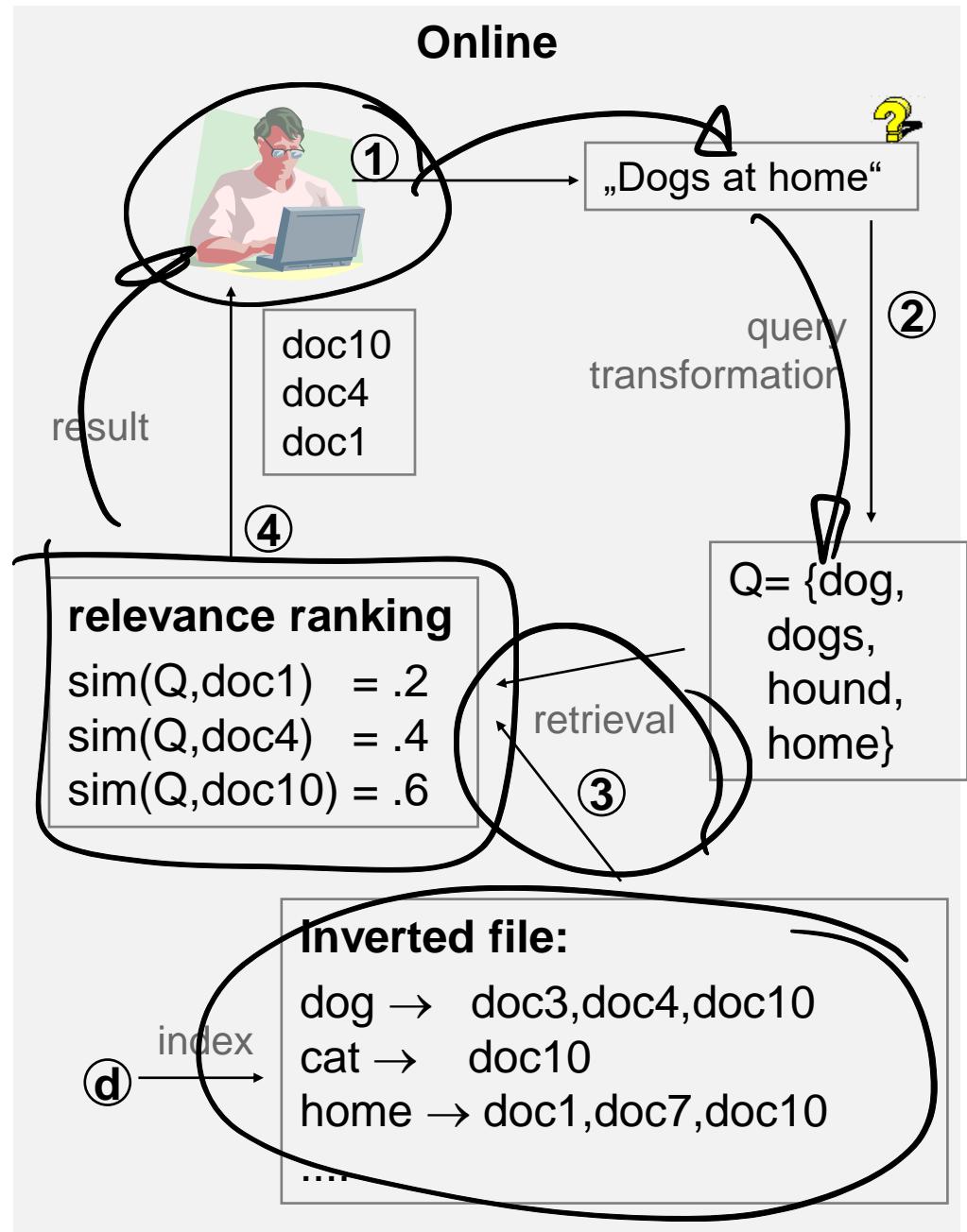
2.3 Feature Extraction

Offline



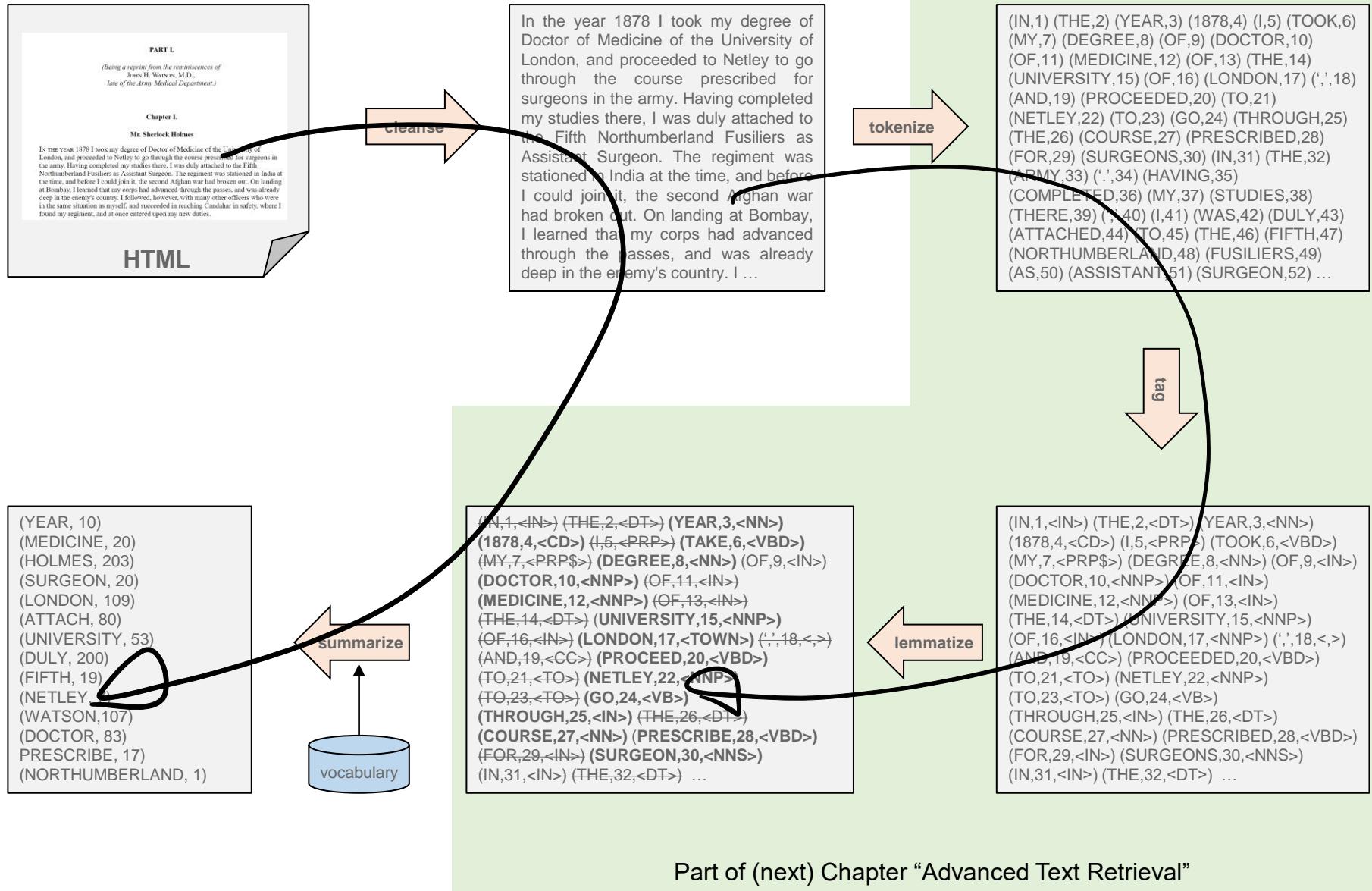
- Text retrieval encompasses two modes:
 - an **offline mode**, that allows us to add documents and to analyze them, and
 - an **online mode**, that retrieves relevant documents for queries given by users
- Obviously, we do not want to apply text search on the native documents. Rather we **extract** so-called **features** which represent characteristic pieces of information about the content of the document. The features also should **support fast retrieval** afterwards.
- In more detail, the following steps occur during the **offline mode**:
 - a) We add a new document (or we find a new document by scanning/crawling)
 - b) Each addition triggers an event to **extract** features and update search indexes
 - c) We **extract** features that best **describe** the **content** and **analyze & reason** on the context and higher-level features
 - d) We **pass the features to an index** that accelerates searches given a query

- In the **online mode** users can search for documents. The query is analyzed similarly to the documents in the offline mode, but often we apply additional processing to correct spelling mistakes or to broaden the search with synonyms. The **retrieval**, finally, is a comparison at the feature level. We assume that two documents that have similar features also are similar in content. Hence, if the **features of the query** are close to the ones of the document, the document is considered a good match.
- In more detail, the following steps occur during the **offline mode**:
 - 1) User enters a query (or speech/ handwriting recognition)
 - 2) We extract features like for the documents, and transform the query as necessary (e.g., spelling mistakes)
 - 3) We use the query features to search the index for document with similar features
 - 4) We rank the documents (retrieval status value, RSV) and return best documents



- Normally, we **do not search through documents with string operations**. Rather, we extract characteristic features that describe the essence of the document in a concise way, and operate on these features only. In this chapter, we first look at lower level features that relate directly to the character sequence. Later on, we extract higher level features, for instance, classifiers, that describe the content with more abstract concepts.
- **Feature extraction comprises of several steps** which we subsequently analyze in more details:
 1. Cleanse document and reduce to sequence of characters
 2. Create tokens from sequence
 3. Tag token stream with additional information
 4. Lemmatization, spell checking, and linguistic transformation (transformation to terms)
 5. Summarize to feature vector (given a vocabulary)
- Note: meta data is often just a special case of text data associated with the document. We will see with Lucene how you can combine text retrieval with predicates over meta data

• Example of Feature Extraction



2.3.1 Step 1: Cleanse Document (with the example of HTML)

- Text documents come in various formats like HTML, PDF, EPUB, or plain text. The initial step is to extract meta information and the sequence of characters that make up the text stream. This may include structural analysis of the document, encoding adjustments, and the identification of relevant information for the feature extraction. We do not want to index control sequences!
- Let us look at a simple example in HTML. The following snippet contains the rough structure of a web page. The first step is to identify which parts contain meaningful information. The header has rich meta information, the body contains the main text parts. Even though HTML is a well-defined standard, extracting information (so-called scraping) requires analysis of the data structure used for the pages. A web search engine simply considers everything.

```
<html>
  <head>
    <title> MMIR - SS01 </title>
    <meta name="keywords"
          content="multimedia, information,
          retrieval, course">
  </head>

  <body>
    ...
  </body>
</html>
```

Header:

Contains meta-information about the document. We can use this information both for adding relevant features as well as cataloguing the document.

Body:

Contains the main content enriched with markups. The flow of the document is not always obvious and may look different on screen than in the file

- **Meta data:** the Web standards provide ways to define meta-information such as:
 - URI of page: (may contain concise key words)
`http://www-dbs.ethz.ch/~mmir/`
 - Title of document: (concise summary of what to expect)
`<title>Multimedia Retrieval - Homepage</title>`
 - Meta information in header section: (enriched information provided by author)
`<meta name="keywords" content="MMIR, information, retrieval,">`
`<meta name="description" content="This will change your life...">`

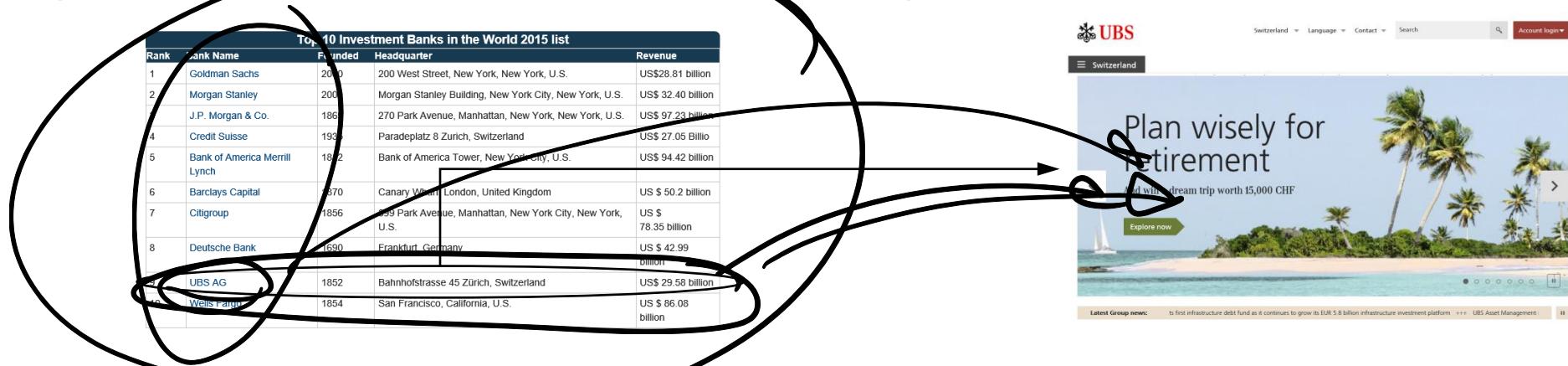
The typical approach is to use the meta data for both the catalogue entry of the document and the text sequence. If we know the context of web pages, we can extract more accurate information.

- **Body Text:** the body subsumes all text blocks and tags them to control presentation. The **flow on the page** must not necessarily follow the order in the **HTML file**, but it's typically a good enough approximation. Some of the tags provide useful additional information on the text pieces:
 - Headlines: `<h1>2. Information Retrieval </h1>`
 - Emphasized: `Please read carefully!`
 or `<i>Information Retrieval</i>`

A typical approach is to add meta information into the text stream based on the HTML tags. For instance, we could assign higher weights to bold-faced terms.

- **Encoding:** most formats provide escape sequences or special characters, that need to be normalized. Furthermore, each document may use a different encoding which may lead to difficulties when searching for terms due to differences in representations
 - `&nbsp` -> space, `&uuml` -> ü
 - Transformation to Unicode, ASCII or other character set

- Web pages contain links. How do we handle them best? They describe relationships between documents and can add to the description of the current document. But more importantly, they also describe the referenced document. As authors of web pages keep link texts rather small, the set of keywords in links is an excellent source for additional keywords for the referenced document.



- Embedded objects (image, plug-ins).

`<IMG SRC="img/MeAndMyCar.jpeg"`

`ALT="picture of me in front of my car">`

- Links to external references:

``

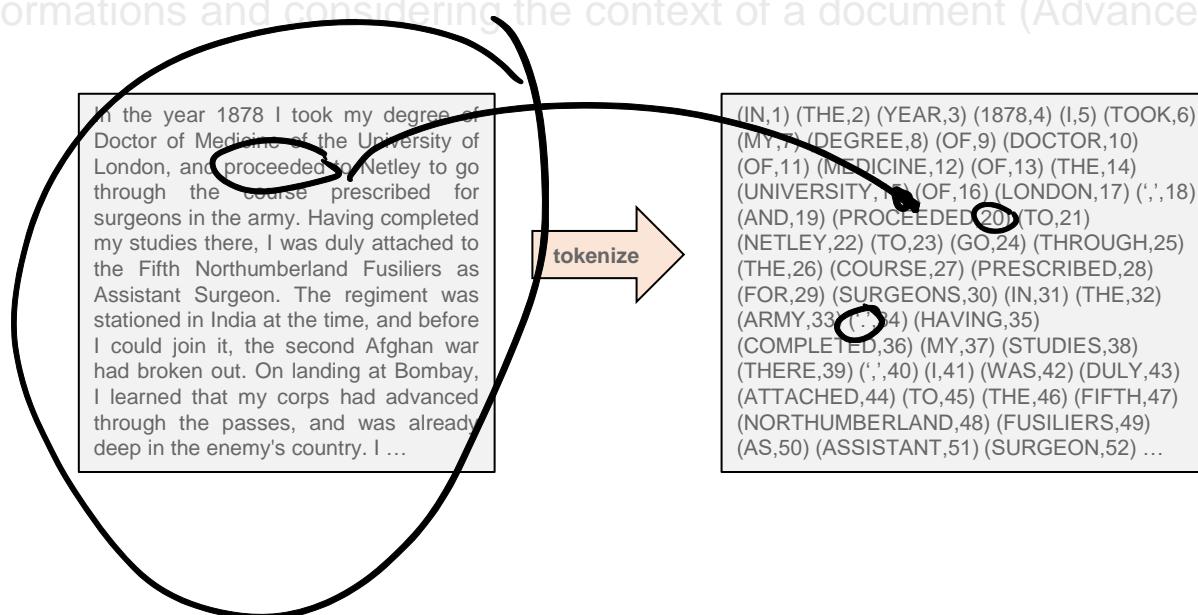
`read this important note`

- **Approach:** Usually, the link text is associated with both the embedding and the linked document.

Typically, we weigh keywords much higher for the referenced document. Be aware of the effectiveness of this approach, e.g., when considering click baits (promises much more than the referenced documents reveal) or navigational hints ("click here", "back to main page"). We will address this in the Web Retrieval chapter in more details.

2.3.2 Step 2-4: Create Tokens

- In this chapter, we merge the steps 2-4 and only apply the basic method of extracting consecutive character sequences (tokens). These tokens correspond to the natural definition of words in text documents. We will go into more details of tokenization in the next chapter (Advanced Text Retrieval) and look at alternative models of extracting tokens (and how to deal with special cases)
- For the remainder of this chapter, we use these tokens directly as terms that describe the content of the document. We will refine this with more sophisticated methods applying linguistic transformations and considering the context of a document (Advanced Text Retrieval).



2.3.3 Step 5: Summarize to Feature Vectors

- Before we create a feature vector, we first define the vocabulary and decide how to statistically summarize the term information.
- Vocabulary:** how many different terms does a collection of documents contain? Church and Gale gave a very good and rough estimator: the number of distinct terms is about the square root of the number of tokens in the entire collection. But not all of these terms are equally important for the retrieval task. So how can we find the most important ones?
 - We usually normalize terms before we add them to the vocabulary (but this is not necessary). As discussed in the previous section, we may end up with near-stems or real stems of the words. Normalization not only reduces the size of vocabulary but it also merges different terms with (mostly) the same meaning. For instance:
 - we can consider the set {cat, cats, cat's, cat's'} as 4 individual terms or as a single term
 - we can treat a synset as one term or each constituent of the synset as an individual term
 - Regardless of the chosen method to extract and normalize terms, we want to eliminate terms that do not help much describing the content of the document. For instance, the term 'it' is used in almost every English text and bears little information about the content. So we may want to ignore those so-called **stop words**; here some examples for English:

i me my myself we our ours ourselves you your yours yourself yourselves he him his himself she her hers herself it its itself they them their theirs themselves what which who whom this that these those am is are was were be been being have has had having do does did doing a an the and but if or because as until while of at by for with about against between into through during before above below to from up down in out on off over under again further then once here there when where why how all any both each few more most other some such no nor not only own same so than too very s t can will just don should now d ll m o re ve y ain aren couldn didn doesn hadn hasn haven isn ma mightn mustn needn shan shouldn wasn weren won wouldn



- Stop word elimination is very common but bears some risks if not done carefully. In the example before, we stated that “it” is not meaningful to distinguish English texts. But consider this:
 - Stephen King wrote a book “It” – We never will find this book if we eliminate ‘it’ as a stop word
 - If we write IT we actually mean information technology – even though it looks like our ‘it’, the big IT is a homonym with a very distinct meaning
 - What do you get if you search the web for ‘it’?
- The other extreme case are seldom terms (or bigrams, n-grams) that only appear once in the entire collection. This multimedia retrieval course is the only one containing the bigram **endoplasmic reticulum**. Is it worth to index this bigram? Is any student ever going to search for this in a computer science collection? If this is unlikely, why bother with such terms.
- A final issue are **spelling mistakes**. Britney, Britni, Bridney, Britnei all appear similar but are different terms for our retrieval system. Misspellings not only blows up our vocabulary (consider all spelling mistakes ever done by any person!), but they also make it impossible to retrieve the content by the correct spelling. On the other side, all of the names given before do also exist (maybe in some cases the parents misspelled the name on the form)

- A pragmatic approach to control vocabulary size is based on **Zipf's law**. Let N be the total number of term occurrences (tokens) in the collection and M be the number of distinct terms in the vocabulary. We already used the **term frequency** $tf(t)$ to denote the number of occurrences of term t . Now, let us order all terms by **decreasing term frequencies** and assign $rank(t)$ to term t based on that order. The central theorem of Zip's law is that the **probability** p_r of randomly selecting the term t with $rank(t) = r$ from the collection is

$$p_r = \frac{c}{r} = \frac{tf(t)}{N} \quad \text{for the term } t \text{ with } rank(t) = r. \quad c \text{ is a constant depending only on } M$$

In other words, we always get the **same constant value** $c \cdot N$ if we multiply the rank of a term with its **term frequency**. Or we can estimate the rank of a term t as: $rank(t) = c \cdot \frac{N}{tf(t)}$. We can easily compute c as a function of M as follows:

$$1 = \sum_{r=1}^M p_r = \sum_{r=1}^M \frac{c}{r} = c \cdot \sum_{r=1}^M \frac{1}{r} \quad \rightarrow \quad c = \frac{1}{\sum_{r=1}^M \frac{1}{r}} \approx \frac{1}{0.5772 + \ln M}$$

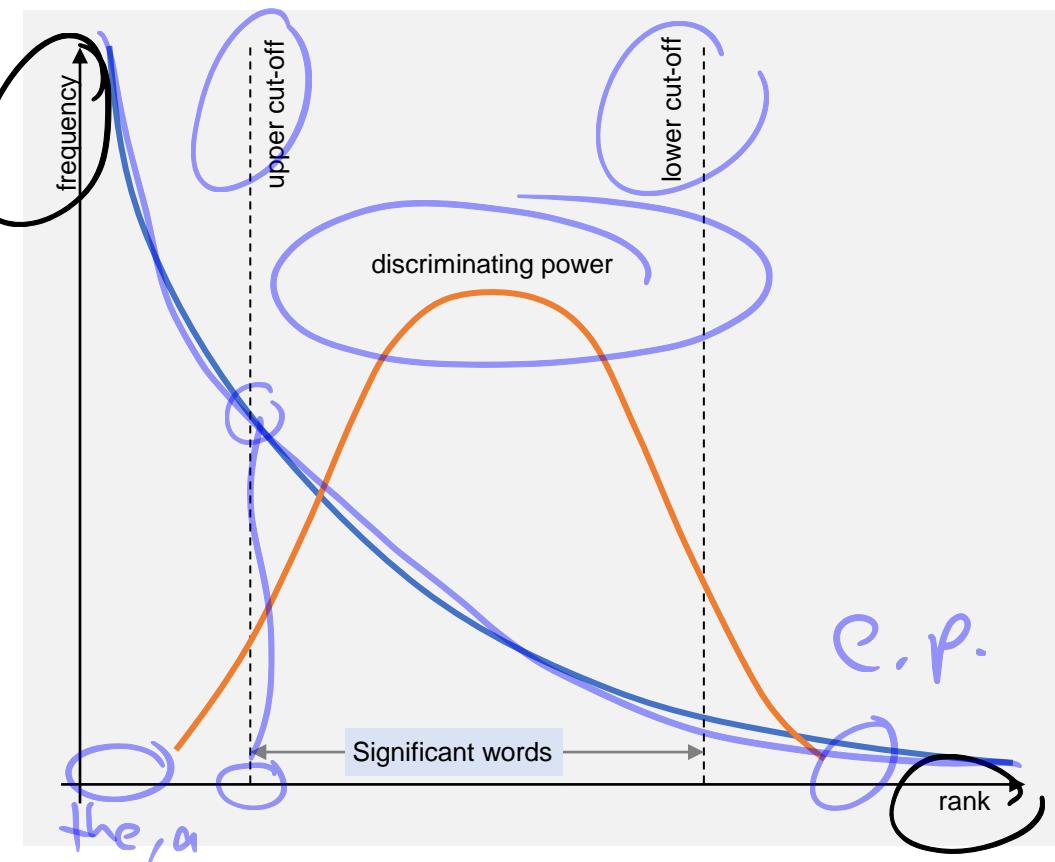
With this we get a simple lookup table for c given the number M of distinct terms:

M	5'000	10'000	50'000	100'000
c	0.11	0.10	0.09	0.08

- The right hand figure shows the Zipf distribution. As discussed, the most frequent words (above the upper cut-off line) bear little meaning as they occur in almost every text. The least frequent words (below the lower cut-off) appear too seldom to be used in queries and only discriminate a few documents. The range of significant words lies in between the lower and upper cut-off.

- Originally, the idea was to define the cut-off thresholds and eliminate the words outside the indicated range. This would save memory and speed up search. This has become irrelevant.

- Today, the typical approach is to eliminate only stop words from a short well-maintained list, or to keep even all terms as the additional (storage) overhead is minimal. On the other side, we can use Zipf's law to weigh the terms. With these weights, we can express how well a term can distinguish between relevant and non-relevant documents. The figure above indicates that power of discrimination with the red plot. Note that even though the very rare terms are directly pointing to the relevant documents, they are also rarely used in queries and, hence, their expected discrimination power is low. The best terms are those, that divide documents clearly (contain term, do not contain term) and are frequently used in queries.

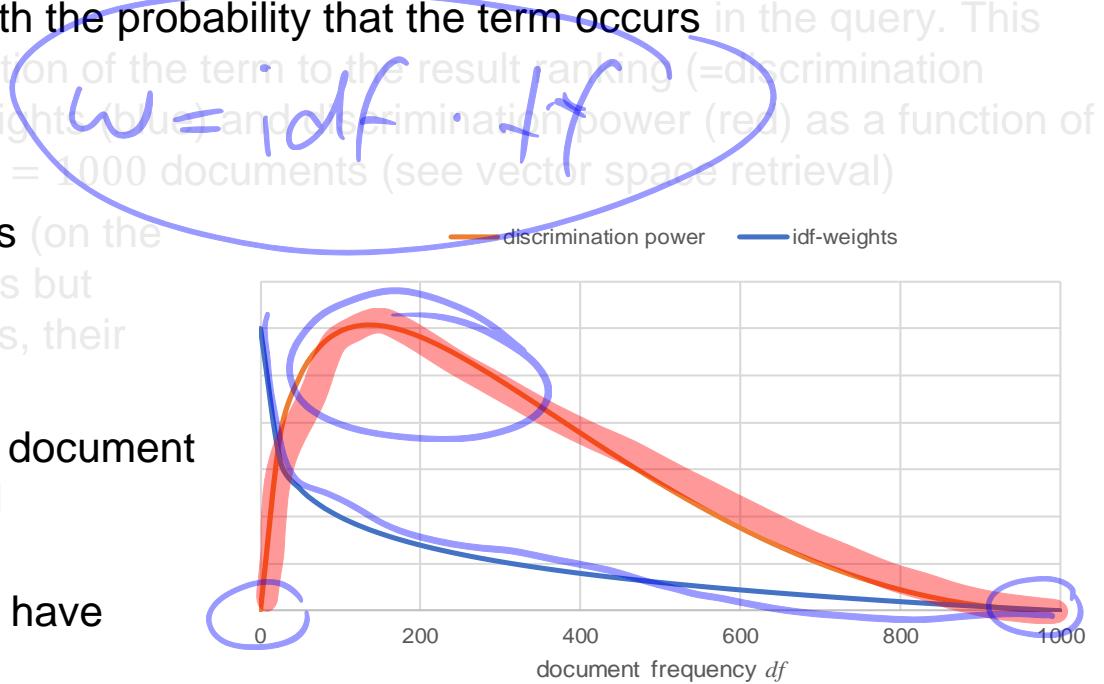


- **Discrimination power:** in vector space retrieval, we use the so-called **inverse document frequency** to define weights on terms that correspond directly to their discrimination power. Instead of counting the total number of occurrences as in the term frequency $tf(t)$, the document frequency $df(t)$ counts in how many documents the term t appears at least once. Let N be the number of documents in the collection. The inverse document frequency $idf(t)$ is then given as (note that there are many similar definitions for $idf(t)$):

$$idf(t) = \log \frac{N + 1}{df(t) + 1} = \log(N + 1) - \log(df(t) + 1)$$

The inverse document frequency describes the weight of a term both in the document description as well as in the query description. We can estimate the discrimination power of a term t by multiplying the squared $idf(t)$ -value with the probability that the term occurs in the query. This values estimates the expected contribution of the term to the result ranking (=discrimination power). The figure below shows idf -weights (blue) and discrimination power (red) as a function of the document frequency df and with $N = 1000$ documents (see vector space retrieval)

- Terms with low document frequencies (on the left side) have the highest idf -weights but as they also seldom appear in queries, their discrimination power is low
- On the right side, the terms with high document frequency have both low weights and discrimination power.
- The terms around $df = 100 = 0.1 \cdot N$ have the highest discrimination power.



- The **discrimination method** provides an alternative to the idf -weights. In essence, we want to measure how much a term is able to discriminate the document collection, or from a different angle: if we remove the term from the collection, how much more similar do the documents become without that term. Let $0 \leq sim(D_i, D_j) \leq 1$ denote the **similarity** between two documents D_i and D_j where 0 means the documents are totally dissimilar and 1 means they are identical.

- In a collection with N documents, compute the **centroid document C** as the document that

contains all M terms with mean frequency over the collection. If $tf(D_i, t_j)$ is the term frequency of term t_j in document D_i , then

$$tf(C, t_j) = \frac{1}{N} \cdot \sum_{i=1}^N tf(D_i, t_j) \quad \text{for } \forall j: 1 \leq j \leq M$$

- We define the **density** of the collection as the sum of all similarities between documents and

their centroid C :

$$Q = \sum_{i=1}^N sim(D_i, C)$$

- Now assume we remove the term t from the collection. We can compute the density for this

modified collection and then define the discrimination power of term t as:

$$dp(t) = Q_t - Q$$

- If the discrimination value is large, Q_t is larger than Q . Hence, if we remove the term t from the collection, similarities to the centroid become larger. If we add the term again, documents become more distinct from their centroid. In other words, the term t **differentiates the collection and is hence a significant term**. On the other side, if $dp(t)$ is negative, we conclude that Q is larger than Q_t . That means if we remove the term from the collection, documents become more distinct from the centroid. If we add the term again, the documents become more similar to the centroid. In other words, the term is likely “spamming” the collection and has a (very) negative impact on describing the documents. For example, if we add the term “hello” a 1'000 times to each document, they obviously become more similar to each other (and the centroid). Hence, terms with very small $dp(t)$ are not significant (or even harmful) to describe the collection.
- We can now select the most useful terms by ordering them by their decreasing $dp(t)$ -values and cut-off the list if the discrimination value falls below some threshold value.
- Once the vocabulary is fixed, we can describe documents D_i by a feature value d_i . The **set-of words** model is a simple representation that only considers whether a term is present and disregards order of terms, number of occurrences, and proximity between terms. The most simple representation is the set of terms appearing at least once, that is a binary feature vector where dimension j denotes the presence ($= 1$) or absence ($= 0$) of term t_j .

$$d_{i,j} \in \{0,1\}^M, \quad d_{i,j} = \begin{cases} 1 & tf(D_i, t_j) > 0 \\ 0 & tf(D_i, t_j) = 0 \end{cases} \quad \text{or} \quad d_i = \{t_j \mid tf(D_i, t_j) > 0\}$$

The **bag-of-words** model is the more common representation and differs from the set-of-words by keeping multiplicity of terms. The representation is a feature vector over term frequencies

$$d_{i,j} \in \mathbb{N}^M, \quad d_{i,j} = tf(D_i, t_j)$$

2.4 Text Retrieval Models

- In the following sections, we consider different retrieval models and discuss their advantages and disadvantages. We only touch the essential method while there are many more extensions in the literature. We will use the following notations in this chapter:

Notation	Value Range	Description
\mathbb{D}	$\{D_1, \dots, D_N\}$	Collection of N documents
D_i		Representation of a document with $1 \leq i \leq N$
\mathbb{T}	$\{t_1, \dots, t_M\}$	Collection of M terms
t_j		Representation of a term with $1 \leq j \leq M$
\mathbf{d}_i	$\{0,1\}^M, \mathbb{N}^M$, or \mathbb{R}^M	Feature description of document D_i with the j -the dimension describing document with regard to term t_j
\mathbf{A}	$\{0,1\}^{M \times N}, \mathbb{N}^{M \times N}$, or $\mathbb{R}^{M \times N}$	Term-document matrix with $a_{j,i} = tf(D_i, t_j)$, that is rows denote terms and columns denote documents. For instance, the i -th column is $a_{:,i} = \mathbf{d}_i$.
$tf(D_i, t_j)$	\mathbb{N}	Term frequency of term t_j in document D_i , i.e., number of occurrences of term t_j in document D_i
$df(t_j)$	\mathbb{N}	Document frequency of term t_j in the collection \mathbb{D} , i.e., number of documents in \mathbb{D} that contain term t_j at least once
$idf(t_j)$	\mathbb{R}	Inverse document frequency of term t_j given by $idf(t_j) = \log(N + 1) - \log(df(t_j) + 1)$
Q		Representation of a query
\mathbf{q}	$\{0,1\}^M, \mathbb{N}^M$, or \mathbb{R}^M	Feature description of query Q with the j -the dimension describing query with regard to term t_j
$sim(Q, D_i)$	[0,1]	Similarity between query Q and document D_i . 0 means dissimilar, 1 means identical

2.4.1 Standard Boolean Model

- The standard Boolean model is the classical text retrieval method introduced in the 1970s. Given the limited capabilities of computing at this time, it was important that we can answer queries by considering only the current data set (tape drives were sequential). Even though more advanced methods were developed, it is still used by many engines and still works fairly well.
- As the name suggests, the model operates on Boolean logic over sets of terms. Documents are represented by sets of words, and queries come from the following grammar:

- $Q = t$ Term t must be present
- $Q = \neg t$ Term t must not be present
- $Q = Q_1 \vee Q_2$ Sub-query q_1 or sub-query q_2 fulfilled
- $Q = Q_1 \wedge Q_2$ Both sub-query q_1 and q_2 fulfilled

- To evaluate such queries, we can transform them into their disjunctive normal form

$$Q = (\tau_{1,1} \wedge \dots \wedge \tau_{1,K_1}) \vee \dots \vee (\tau_{L,1} \wedge \dots \wedge \tau_{L,K_L}) = \bigvee_{l=1}^L \left(\bigwedge_{k=1}^{K_l} \tau_{l,k} \right)$$

with $\tau_{l,k} = t_{j(l,k)}$ or $\tau_{l,k} = \neg t_{j(l,k)}$ ($j(l,k)$ is mapping to the index of the term used in the query)

- For each atomic part $\tau_{l,k}$, we can compute the set $\mathbb{S}_{l,k}$ of documents that contain or do not contain the term:

$$\mathbb{S}_{l,k} = \begin{cases} \{D_i \mid tf(D_i, t_{j(l,k)}) = 1\} & \text{if } \tau_{l,k} = t_{j(l,k)} \\ \{D_i \mid tf(D_i, t_{j(l,k)}) = 0\} & \text{if } \tau_{l,k} = \neg t_{j(l,k)} \end{cases}$$

- The final result \mathbb{Q} is then a combination of intersections and unions over the sets derived from the atomic parts

$$\mathbb{Q} = \bigcup_{l=1}^L \bigcap_{k=1}^{K_l} \mathbb{S}_{l,k} = \bigcup_{l=1}^L \bigcap_{k=1}^{K_l} \begin{cases} \{D_i \mid tf(D_i, t_{j(l,k)}) = 1\} & \text{if } \tau_{l,k} = t_{j(l,k)} \\ \{D_i \mid tf(D_i, t_{j(l,k)}) = 0\} & \text{if } \tau_{l,k} = \neg t_{j(l,k)} \end{cases}$$

- Advantages:** simple model with a clean description of query semantics. Very simple to implement and intuitive for users. Even though the definition of query evaluation is based on sets, we will see later in this chapter that the inverted lists provides a very efficient way to compute the inner intersections of the evaluation (with some restrictions on query structure). The Boolean expression provides an accurate way to define what relevance means.
- Disadvantages:** no (intuitive) control over the size of retrieved documents and a user may get either too few or too many results. For larger result sets, the lack of ranking requires the user to browse through the documents to find the best match. Although the query language is simple, users may find it hard to express a complex information need as a combination of ANDs and ORs. All terms are treated equally, hence, stop words contribute equally to the result as the more significant terms. Retrieval quality is ok but other methods (with equal computational complexity) achieve much better results.

2.4.2 Extended Boolean Model

- The lack of ranking is a huge handicap of the standard Boolean model. The extended versions of the Boolean model overcome this drawback: we consider term weights, use the bag of words model, and apply partial matching capability similar to the vector space retrieval model. The algebra is still Boolean but evaluations return a similarity value rather than a 0/1-view. There are several variants but they all follow a similar concept.
- A document D_i is represented as a vector \mathbf{d}_i with normalized term frequencies:

$$d_{i,j} = \min \left(1, \frac{tf(D_i, t_j) \cdot idf(t_j)}{\alpha} \right) \quad \forall j: 1 \leq j \leq M \quad \text{with } \alpha = \max \left(tf(D_i, t_j) \cdot idf(t_j) \right) \quad (\text{or some other value})$$

Other methods to normalization are possible (like the discrimination value we have seen previously). A query Q follows the same structure as in the standard Boolean model, hence:

$$Q = (\tau_{1,1} \wedge \dots \wedge \tau_{1,K_1}) \vee \dots \vee (\tau_{L,1} \wedge \dots \wedge \tau_{L,K_L}) = \bigvee_{l=1}^L \left(\bigwedge_{k=1}^{K_l} \tau_{l,k} \right)$$

with $\tau_{l,k} = t_{j(l,k)}$ or $\tau_{l,k} = \neg t_{j(l,k)}$ ($j(l,k)$ is mapping to the index of the term used in the query)

- For each atomic part $\tau_{l,k}$, we can compute the similarity value $sim(Q = \tau_{l,k}, D_i)$ for a document D_i :

$$sim(Q = \tau_{l,k}, D_i) = \begin{cases} d_{i,j(l,k)} & \text{if } \tau_{l,k} = t_{j(l,k)} \\ 1 - d_{i,j(l,k)} & \text{if } \tau_{l,k} = \neg t_{j(l,k)} \end{cases}$$

- There are several variants that calculate the AND and OR operators.

- **Fuzzy Algebraic:** (only works for two operands)

$$\text{sim}(Q_1 \wedge Q_2, D_i) = \text{sim}(Q_1, D_i) \cdot \text{sim}(Q_2, D_i)$$

$$\text{sim}(Q_1 \vee Q_2, D_i) = \text{sim}(Q_1, D_i) + \text{sim}(Q_2, D_i) - \text{sim}(Q_1, D_i) \cdot \text{sim}(Q_2, D_i)$$

- **Fuzzy Set:** (generalization to K sub-queries is straight forward)

$$\text{sim}(Q_1 \wedge Q_2, D_i) = \min\{\text{sim}(Q_1, D_i), \text{sim}(Q_2, D_i)\}$$

$$\text{sim}(Q_1 \vee Q_2, D_i) = \max\{\text{sim}(Q_1, D_i), \text{sim}(Q_2, D_i)\}$$

- **Soft Boolean Operator:** (generalization to K sub-queries is straight forward)

$$\text{sim}(Q_1 \wedge Q_2, D_i) = (1 - \alpha) \cdot \min\{\text{sim}(Q_1, D_i), \text{sim}(Q_2, D_i)\} + \alpha \cdot \max\{\text{sim}(Q_1, D_i), \text{sim}(Q_2, D_i)\} \quad 0 \leq \alpha \leq 0.5$$

$$\text{sim}(Q_1 \vee Q_2, D_i) = (1 - \beta) \cdot \min\{\text{sim}(Q_1, D_i), \text{sim}(Q_2, D_i)\} + \beta \cdot \max\{\text{sim}(Q_1, D_i), \text{sim}(Q_2, D_i)\} \quad 0.5 \leq \beta \leq 1$$

- **Paice-Model:** order the sub-queries in increasing order of their similarity values for AND operator, and order the sub-queries in decreasing order of their similarity values for OR. r is a constant coefficient:

$$\text{sim}\left(\bigwedge_{k=1}^K Q_k, D_i\right) = \frac{\sum_{k=1}^K r^{k-1} \cdot \text{sim}(Q_k, D_i)}{\sum_{k=1}^K r^{k-1}} \quad \text{with } \forall k, 1 \leq k < K: \text{sim}(Q_k, D_i) \leq \text{sim}(Q_{k+1}, D_i)$$

$$\text{sim}\left(\bigvee_{k=1}^K Q_k, D_i\right) = \frac{\sum_{k=1}^K r^{k-1} \cdot \text{sim}(Q_k, D_i)}{\sum_{k=1}^K r^{k-1}} \quad \text{with } \forall k, 1 \leq k < K: \text{sim}(Q_k, D_i) \geq \text{sim}(Q_{k+1}, D_i)$$

- **P-Norm-Model:**

$$sim\left(\bigwedge_{k=1}^K Q_k, D_i\right) = 1 - \sqrt[p]{\frac{\sum_k (1 - sim(Q_k, D_i))^p}{K}} \quad \text{with } 1 \leq p < \infty$$

$$sim\left(\bigvee_{k=1}^K Q_k, D_i\right) = 1 - \sqrt[p]{\frac{\sum_k sim(Q_k, D_i)^p}{K}}$$

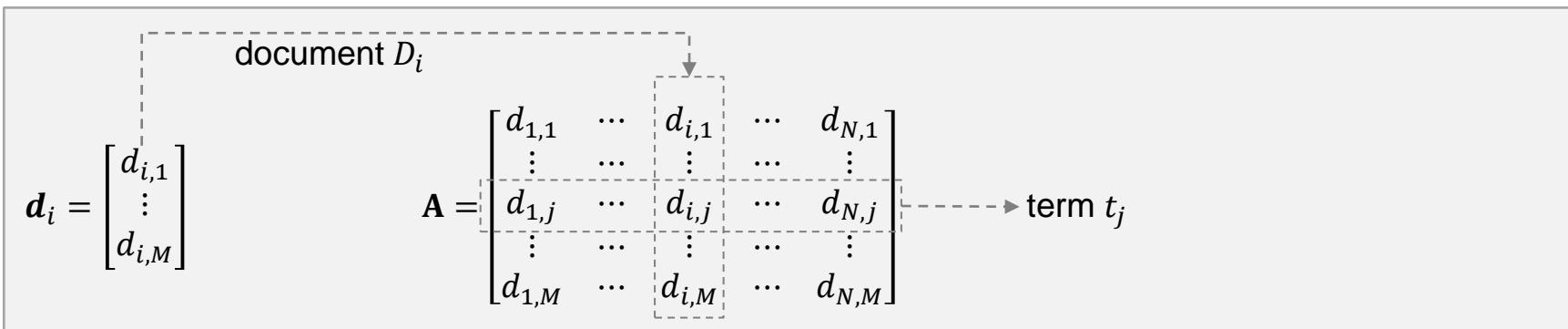
- **Advantages:** simple model with a clean description of query semantics. Very simple to implement and intuitive for users Even though the definition of query evaluation is rather heuristic, performance is quite good. With the inverted lists method, there is a very efficient way to compute the similarity values. In comparison with the standard Boolean model, we now obtain ranked lists and partial matches, i.e., we can control the size of results to be presented back to the user. Terms are treated differently based on their term occurrence and their discrimination power.
- **Disadvantages:** heuristic similarity scores with little intuition why they work well (no theoretic background for the model). Although the query language is simple, users may find it hard to express a complex information need as a combination of ANDs and ORs Retrieval quality is ok but other methods (with equal computational complexity) achieve better results.

2.4.3 Vector Space Retrieval

- The vector space retrieval model is by far the most popular of the classic text retrieval models. It has a clean and simple query structure and offers a very fast computational scheme through inverted lists. In contrast to the Boolean models considered so far, it uses the bag-of-words model both to describe the documents and the queries. In other words, a query is considered as a (mini) document and then used as a reference to find similar documents
- A document D_i is represented as a vector \mathbf{d}_i using weighted term frequencies (we do not normalize the term frequencies as with the extended Boolean models):

$$d_{i,j} = tf(D_i, t_j) \cdot idf(t_j) \quad \forall j: 1 \leq j \leq M$$

- All the vectors \mathbf{d}_i of the collection \mathbb{D} form the so-called term-document-matrix \mathbf{A} with d_i denoting the i -th column of the matrix, i.e., $a_{j,i} = d_{i,j}$ (the switch of indexes is necessary as \mathbf{d}_i is a column vector). A visual representation is as follows:



It follows that the j -th row in \mathbf{A} contains the information about the term t_j .

- Queries are represented as (very sparse) documents. In other words, the user is not required to enter a complex Boolean query but rather provides a few keywords to search for. A query Q is hence represented as a vector q just like all the documents:

$$q_j = tf(Q, t_j) \cdot idf(t_j) \quad \forall j: 1 \leq j \leq M$$

- We can compute similarity values between documents and queries as a function over the M -dimensional vectors. Two popular methods exists:
 - The **inner vector product** uses the dot-product over vectors to calculate similarity values.

$$sim(Q, D_i) = \mathbf{q} \cdot \mathbf{d}_i = \sum_{j=1}^M q_j \cdot d_{i,j}$$

We can also represent all similarity values between documents D_i and the query Q as a **matrix multiplication**:

$$\mathbf{sim}(Q, \mathbb{D}) = \begin{bmatrix} sim(Q, D_1) \\ \vdots \\ sim(Q, D_N) \end{bmatrix} = \mathbf{A}^T \mathbf{q}$$

Note that we only write the above formula for the sake of concise presentation, but we never actually perform matrix multiplications to search for documents. Intuitively, documents are similar to the query if they use the same term as the query (all terms not used in the query have a 0 in q). If the query terms are frequently used, high similarity values result. Further we observe that not all query terms are necessary to obtain non-zero similarities (\rightarrow partial matches)

- The second measure calculates the **cosine** of the angle between the query vector and the document vector to calculate similarity values.

$$sim(Q, D_i) = \frac{\mathbf{q} \cdot \mathbf{d}_i}{\|\mathbf{q}\| \cdot \|\mathbf{d}_i\|} = \frac{\sum_{j=1}^M q_j \cdot d_{i,j}}{\sqrt{\sum_{j=1}^M q_j^2} \cdot \sqrt{\sum_{j=1}^M d_{i,j}^2}}$$

Again, a matrix multiplication leads to all similarity values:

$$sim(Q, \mathbb{D}) = \begin{bmatrix} sim(Q, D_1) \\ \vdots \\ sim(Q, D_N) \end{bmatrix} = \mathbf{L} \mathbf{A}^T \mathbf{q}' \quad \text{with } \mathbf{L} \in \mathbb{R}^{N \times N} = \begin{bmatrix} \frac{1}{\|\mathbf{d}_1\|} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \frac{1}{\|\mathbf{d}_N\|} \end{bmatrix} \quad \text{and } \mathbf{q}' = \frac{\mathbf{q}}{\|\mathbf{q}\|}$$

As before, we only write the above formula for the sake of concise presentation, but we never actually perform matrix multiplications to search for documents. Intuitively, documents are similar to the query if their vectors point to the same direction as the query vector. The number of terms and the weights only play a role to define the direction but the length of the vectors is irrelevant. This provides an equal chance for small and large documents to obtain a high similarity value.

- **Example:** we consider a very simple collection of three documents to observe how the method works. The documents and the query are as follows:

D_1	Shipment of gold damaged in a fire
D_2	Delivery of silver arrived in a silver truck
D_3	Shipment of gold arrived in a truck
Q	gold silver truck

- We can extract terms and determine document frequencies and inverse document frequencies.
The document and query are then represented as vectors ($N = 3, M = 11$):

j	Term t_j	$df(t_j)$	$idf(t_j)$
1	a	3	0
2	arrived	2	.176
3	damaged	1	.477
4	delivery	1	.477
5	fire	1	.477
6	gold	2	.176
7	in	3	0
8	of	3	0
9	silver	1	.477
10	shipment	2	.176
11	truck	2	.176

To simplify, we use: $idf(t_j) = \log(N) - \log(df(t_j))$

with inner
vector product

$$\text{sim}(\mathbf{Q}, \mathbb{D}) = \begin{bmatrix} .031 \\ .486 \\ .062 \end{bmatrix}$$



$$D_2 > D_3 > D_1$$

- **Observations:** the term-document matrix is usually very sparse, that is a single document only contains a small subset of all possible terms. We also note that we only need to consider the query terms for evaluation; all other terms are eliminated due to the 0-value in q . On the other side, a document does not have to contain all query terms to be relevant. In the example before, none of the documents contained all terms. To express such a partial match query with Boolean operators would quickly lead to quite complicated expressions. In the example before, the partial match query in Boolean terms is: (gold AND silver AND truck) OR (gold AND silver) OR (gold AND truck) OR (silver AND truck) OR gold OR silver OR truck.
- **Advantages:** extremely simple and intuitive query model. Very simple to implement and very fast to calculate. Performance is better than with Boolean models and can compete with the best retrieval methods. The model naturally includes partial match queries and documents do not have to contain all query terms to obtain high similarity values.
- **Disadvantages:** heuristic similarity scores with little intuition why they work well (no theoretic background for the model). The similarity measures are not robust and can be biased by authors (spamming of terms). Main assumption of retrieval model is independence of terms which may not hold true in typical scenarios (see synonyms and homonyms). There are several extensions that address this latter aspect.

2.4.4 Probabilistic Retrieval

- The biggest criticism for the models so far is the heuristic approach they take. The methods work and perform well, but there is no foundation to prove correctness. Probabilistic retrieval is a formal approach based on the probability $P(R|D_i)$ that a document D_i is relevant for a query Q and the probability $P(NR|D_i) = 1 - P(R|D_i)$ that a document D_i is not relevant for a query Q . The similarity value is defined as:

$$sim(Q, D_i) = \frac{P(R|D_i)}{P(NR|D_i)} = \frac{P(R|D_i)}{1 - P(R|D_i)}$$

- The **Binary Independence Model (BIR)** is a simple technique based on a few assumptions to compute the conditional probabilities above. The assumptions are
 - Term frequency does not matter (we use the set-of-words model for documents)
 - Terms are independent of each other (all models so far made the same assumptions)
 - Terms that are not part of the query do not impact the ranking (if a term does not appear in the query, we assume that it is equally distributed in the relevant and the non-relevant documents)

With these assumptions, we now compute the above similarity function. As a first step, we use Bayes' theorem on the definition above:

$$sim(Q, D_i) = \frac{P(R|D_i)}{P(NR|D_i)} = \frac{P(D_i|R) \cdot P(R)}{P(D_i|NR) \cdot P(NR)}$$

We can interpret these new probabilities as follows: $P(R)$ and $P(NR)$ are the probabilities that a randomly selected document is relevant and not relevant, respectively. $P(D_i|R)$ and $P(D_i|NR)$ are the probabilities that D_i is among the relevant and among the non-relevant documents, respectively.

- We now use the assumption that documents are binary vectors and that terms are independent of each other:

**Assumption 1:
Documents are
binary vectors**

**Assumption 2: Terms
are independent**

**Assumption 1: Documents
are binary vectors**

$$\begin{aligned}
 P(D_i|R) &= P(d_i|R) = \prod_{j=1}^M P(d_{i,j}|R) = \prod_{\forall j: d_{i,j}=1} P(d_{i,j}=1|R) \cdot \prod_{\forall j: d_{i,j}=0} P(d_{i,j}=0|R) \\
 P(D_i|NR) &= P(d_i|NR) = \prod_{j=1}^M P(d_{i,j}|NR) = \prod_{\forall j: d_{i,j}=1} P(d_{i,j}=1|NR) \cdot \prod_{\forall j: d_{i,j}=0} P(d_{i,j}=0|NR)
 \end{aligned}$$

- We introduce a short notation for the conditional probabilities on the right most side of the formula above. Let $r_j = P(d_{i,j} = 1|R)$ denote the probability that a relevant document has the term t_j (i.e., $d_{i,j} = 1$). Further, let $n_j = P(d_{i,j} = 1|NR)$ denote the probability that a not relevant document has the term t_j (i.e., $d_{i,j} = 1$). With that we can write the similarity value as:

$$sim(Q, D_i) = \frac{P(R)}{P(NR)} \cdot \prod_{\forall j: d_{i,j}=1} \frac{r_j}{n_j} \cdot \prod_{\forall j: d_{i,j}=0} \frac{1-r_j}{1-n_j} \rightarrow sim(Q, D_i) \sim \prod_{\forall j: d_{i,j}=1} \frac{r_j}{n_j} \cdot \prod_{\forall j: d_{i,j}=0} \frac{1-r_j}{1-n_j}$$

Note that we do not need to compute $P(R)$ and $P(NR)$ as they are depending only on the query but do not change the order of documents D_i by their similarity values. Hence, the right formula above is a further simplification that yields the same ranking for documents as the left formula.

- We finally use the third assumption that $r_j = n_j$ if the term t_j does not occur in the query (the term occurs equally likely in the set of relevant and non-relevant documents). This means that for all $q_j = 0$, the ratios $\frac{r_j}{n_j}$ and $\frac{1-r_j}{1-n_j}$ are 1 and we can eliminate them from the calculations:

Assumption 3: non-query terms do not impact result

$$\text{sim}(Q, D_i) \sim \prod_{\forall j: d_{i,j}=1} \frac{r_j}{n_j} \cdot \prod_{\forall j: d_{i,j}=0} \frac{1-r_j}{1-n_j} = \prod_{\forall j: d_{i,j}=1, q_j=1} \frac{r_j}{n_j} \cdot \prod_{\forall j: d_{i,j}=0, q_j=1} \frac{1-r_j}{1-n_j}$$

We drop the condition $d_{i,j} = 1$ in the second product and must compensate in the first product:

$$\text{sim}(Q, D_i) \sim \prod_{\forall j: d_{i,j}=1, q_j=1} \frac{r_j \cdot (1-n_j)}{n_j \cdot (1-r_j)} \cdot \prod_{\forall j: q_j=1} \frac{1-r_j}{1-n_j}$$

Next, we eliminate terms that only depend on the query and do not change the ordering:

$$\text{sim}(Q, D_i) \sim \prod_{\forall j: d_{i,j}=1, q_j=1} \frac{r_j \cdot (1-n_j)}{n_j \cdot (1-r_j)}$$

We finally obtain a very simple similarity function as a sum over c_j -values. Note that we only need to compute c_j for query terms, that is for a very small number of terms.

$$\text{sim}(Q, D_i) \sim \sum_{\forall j: d_{i,j}=1, q_j=1} c_j \quad \text{with } c_j = \log \frac{r_j \cdot (1-n_j)}{n_j \cdot (1-r_j)}$$

- Computing the c_j values: recall that $r_j = P(d_{i,j} = 1|R)$ denotes the probability that a relevant document contains the term t_j . Similarly, $n_j = P(d_{i,j} = 1|NR)$ denotes the probability that a non-relevant document contains the term t_j . To obtain estimates for these probabilities, we ask the user to rate some of the retrieved documents. The more feedback we gather, the better our estimates become. In more details:

- **Initial step:** without any samples, we assume that query terms are more likely to occur in relevant documents while they appear in non-relevant documents according to their document frequency. We use the following estimates for the initial step to compute the c_j

$$r_j = 0.5, \quad n_j = \frac{df(t_j)}{N} \quad \forall j: q_j = 1$$

- **Feedback step:** although the initial values are a heuristic, we only use them to generate a first result set. We then ask the user to rate the K retrieved documents and annotate them with relevant (R) and not relevant (NR). Let L be the number of documents that the user marked as relevant. Further let k_j be the number of retrieved documents that contain the term t_j (that is the document frequency of t_j over the set of retrieved documents), and let l_j be the number of retrieved and relevant documents that contain the term t_j (that is the document frequency of t_j over the set of retrieved and relevant documents). With that, we can estimate new values for r_j and n_j as follows:

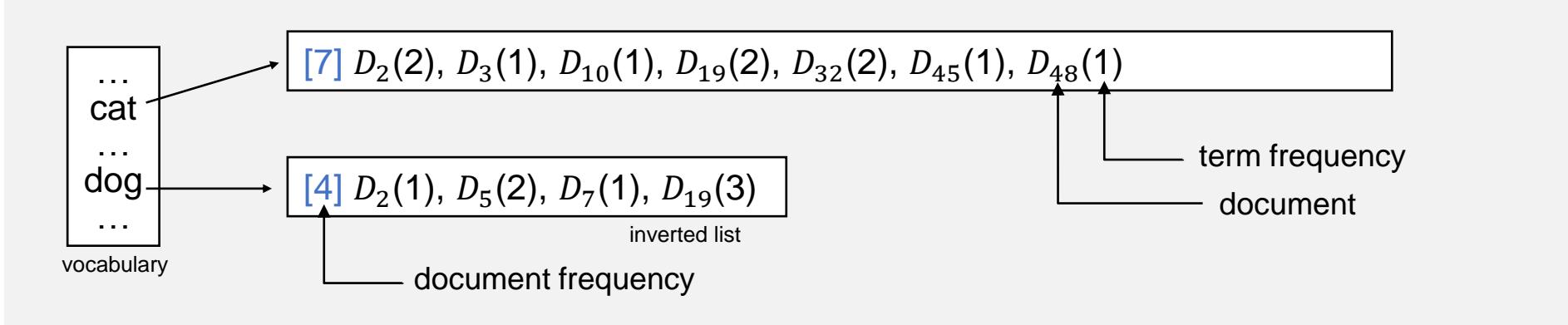
$$r_j = \frac{l_j + 0.5}{L + 1}, \quad n_j = \frac{k_j - l_j + 0.5}{K - L + 1} \quad \forall j: q_j = 1$$

We use the values 0.5 and 1 in the formula above to prevent numerical issues (0-divisions).

- **Advantages:** the BIR model provides a probabilistic foundation based on simple assumptions to define similarity values. The ranking of documents is based on their probability of being relevant for the query. Again, we only require query terms for the calculations of similarity values and, with the inverted lists, we have a very efficient evaluation method at hand. The method provides very good performance, especially after a few feedback steps. It also supports partial match queries, i.e., not all query terms must occur in relevant documents.
- **Disadvantages:** the simple assumptions do not always hold true. Like discussed in the vector space model, term independence does not apply generally. There are more sophisticated probabilistic models that deal with term dependence, but often come with additional computational overhead. Finally, we note that the ranking of documents does neither take term frequencies nor the discrimination power of terms into account.

2.5 Indexing Structures

- With all retrieval models considered so far, we have observed that **ranking** (or selection of an answer in Boolean models) **only depends on query terms**. In addition, if the terms have high discrimination value they are likely to appear in only a few documents. In this section, we look at inverted lists as a simple retrieval model, and apply it to SQL databases for a fast and efficient implementation of text retrieval.
- The term-document matrix is **very sparse**. We expect that **documents only use a small subset of the existing vocabulary**, and many terms in the vocabulary occur only in very few documents. Instead of storing the full matrix, **we keep condensed rows for each term**. For example, we have two terms “dog” and “cat” which appear in some document. In addition, we want to keep track of term frequencies in the documents to apply one of the more sophisticated ranking function. A typical inverted list looks something like this:



- **Application to standard Boolean model:** we can calculate the result with set operations over the atomic parts of the query (must contain term, or must not contain term). The inverted lists provide the sets for the atomic parts “must contain terms”, and, with some restrictions, we can also use them for “must not contain terms”. For example:
 - $Q = \text{"cat"} \text{ AND } \text{"dog"}$
 - $\mathbb{S}_{\text{cat}} = \{D_2, D_3, D_{10}, D_{19}, D_{32}, D_{45}, D_{48}\}$, $\mathbb{S}_{\text{dog}} = \{D_2, D_5, D_7, D_{19}\}$
 - $\mathbb{Q} = \mathbb{S}_{\text{cat}} \cap \mathbb{S}_{\text{dog}} = \{D_2, D_{19}\}$
 - $Q = \text{"cat"} \text{ AND } (\text{NOT } \text{"dog"})$
 - $\mathbb{S}_{\text{cat}} = \{D_2, D_3, D_{10}, D_{19}, D_{32}, D_{45}, D_{48}\}$, $\mathbb{S}_{\text{dog}} = \{D_2, D_5, D_7, D_{19}\}$
 - $\mathbb{Q} = \mathbb{S}_{\text{cat}} - \mathbb{S}_{\text{dog}} = \{D_3, D_{10}, D_{32}, D_{45}, D_{48}\}$
 - More generally, NOT-clauses are only allowed within AND-clauses (translates into minus set-operation), but not in OR-clauses. A query like: “cat” OR (NOT “dog”) cannot be answered with only the inverted lists; in addition, such a query is not really meaningful. So the restriction is hardly relevant for users.
 - To accelerate the set operations, we sort the inverted lists by increasing document frequencies. This way the intermediate results sets are smaller.
- **Retrieval Models with ranking** all the models with ranking that we considered so far, have a partial match capability. In other words, we must retrieve all documents that contain at least one query term and then evaluate the similarity values only for these retrieved documents. For example:
 - $Q = \text{"cat dog"}$ (vector space retrieval, probabilistic retrieval)
 - $Q = \text{"cat AND dog"}$ $Q = \text{"cat AND (NOT dog)"}$ (extended Boolean model)
 - $\mathbb{S} = \mathbb{S}_{\text{cat}} \cup \mathbb{S}_{\text{dog}} = \{D_2, D_3, D_5, D_7, D_{10}, D_{19}, D_{32}, D_{45}, D_{48}\}$

- The typical implementation stores the inverted lists as individual files. But we can also efficiently implement inverted lists in a SQL database and exploit other features that a database provides (proven storage, transaction management, high availability, disaster recovery, ...). For the implementation, we need: 1) a (document) collection, 2) the vocabulary, 3) and the inverted list (here: table Index). In addition, we require a (temporary) query table to simplify SQL queries.

Collection

docid	doc_name	date	dateline
1	WSJ870323-0180	3/23/87	Turin, Italy
2	WSJ870323-0161	3/23/87	Du Pont Company, Wilmington, DE

Index

doc_id	term	tf
1	commercial 1	
1	vehicle	1
1	sales	2
1	italy	1
1	february	1
1	year	1
1	according	1
...		
2	krol	2
2	president	2
2	diversified	1
2	company	1
2	succeeding 1	
2	dale	1
2	products	2
...		

Vocabulary

term	idf
according	0.9031
commercial	1.3802
company	0.6021
dale	2.3856
diversified	2.5798
february	1.4472
italy	1.9231
krol	4.2768
president	0.6990
products	0.9542
sales	1.0000
succeeding	2.6107
vehicle	1.8808
year	0.4771

Query

term	tf
vehicle	1
sales	1
italy	1

- Evaluation of a Boolean Query

- Option 1: no Query table
 $Q = \text{„vehicle sales italy“}$

```
SELECT a.DocID
  FROM Index a,Index b,Index c
 WHERE a.Term='vehicle' AND
       b.Term='sales' AND
       c.Term='italy' AND
       a.DocID=b.DocID AND
       a.DocID=c.DocID;
```

- Option 2: with Query table
 $Q = \text{„vehicle sales italy“}$

```
DELETE FROM Query;
INSERT INTO Query
    VALUES ('vehicle',1);
INSERT INTO Query
    VALUES ('sales',1);
INSERT INTO Query
    VALUES ('italy',1);
```

```
SELECT i.DocID
  FROM Index i, Query q
 WHERE i.Term=q.Term
 GROUP BY i.DocID
 HAVING COUNT(i.Term)=
        (SELECT COUNT(*) FROM QUERY)
```

- Evaluation with Vector Space Retrieval
 - Example: inner vector product
 $Q = \text{„vehicle sales italy“}$

```
DELETE FROM Query;
INSERT INTO Query
    VALUES ('vehicle',1);
INSERT INTO Query
    VALUES ('sales',1);
INSERT INTO Query
    VALUES ('italy',1);

SELECT i.DocID, SUM(q.tf * t.idf * i.tf * t.idf)
    FROM Query q, Index i, Term t
    WHERE q.Term=t.Term AND
        i.Term=t.Term
    GROUP BY i.DocID
    ORDER BY 2 DESC;
```

2.6 Lucene - Open Source Text Search

- Apache hosts several projects to provide easy to use yet powerful text and web retrieval. All of them are based on the core engine called *Lucene*. In addition, third-party libraries enrich Lucene with additional content extractor and analyzers.
 - Lucene: core retrieval library for both analysis of documents and searching
 - Apache Tika: parsers and extractors for various file formats
 - Nutch: open source web search engine with scalable, distributed crawlers and a Tomcat web application to search through the content
 - Solr: open source enterprise search engine for a rich set of file formats
 - Elasticsearch: an enterprise search server
- In this chapter, we look at:
 - how Lucene analyzes documents
 - how Lucene ranks documents
 - how to use Lucene in own applications
- Note: this is not meant to be a complete overview of Lucene. Refer to the online documentation or to books such as “Lucene in Action” to get more details



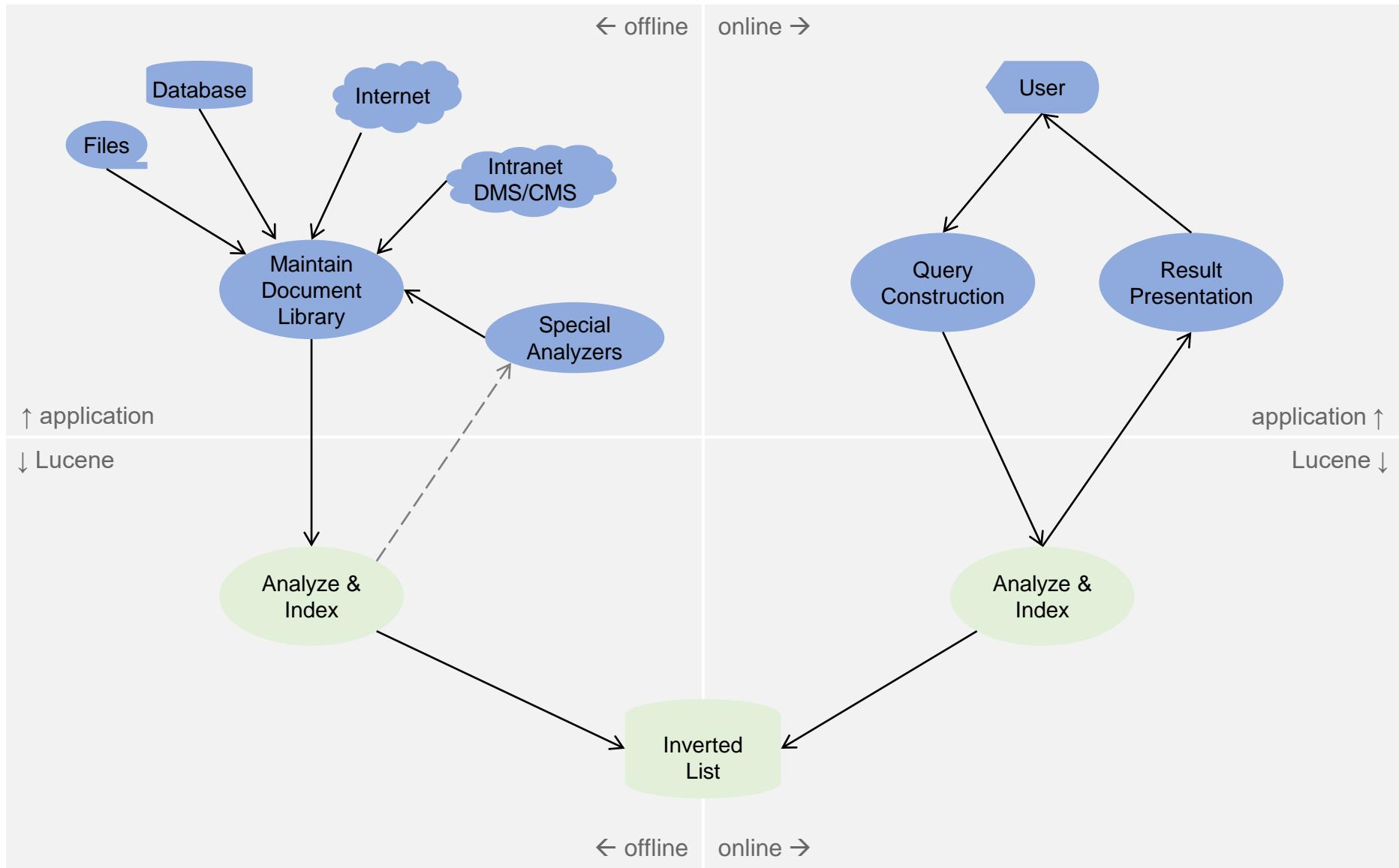
2.6.1 History of Lucene

- Lucene started as a SourceForge project and joined the Apache Jakarta family in 2001. Original author was Doug Cutting. Since 2005, Lucene is a top-level Apache project with many sub-projects. Some of them, namely Nutch and Tika, have become independent Apache projects.
- Main versions introduced (selected versions):
 - 1.01b (July 2001): last SourceForge release
 - 2.0 (May 2006): clean up of code, removed deprecated methods
 - 3.0 (November 2009): cleanup and migration to Java 1.5 (generics, var args)
3.6 is latest build released on July, 2012
 - 4.0 (August 2012): speedup of indexing and retrieval
 - 5.0 (February 2015): index safety, many adjustments on the API
 - 6.0 (April 2016): Java 8, classification, spatial module update
 - 7.0 (September 2017): Java 9 and support of Jigsaw modularization
 - 7.5 (September 2018): Integration of OpenNLP
 - 8.0 (March 2019): Faster custom scores
- Lucene implementations
 - Java (original), C++ (CLucene), .NET (Lucene.NET), C (Lucene4c), Objective-C (LuceneKit), Python (PyLucene), PHP 5 (Zend), Perl (Plucene), Delphi (MUTIS), JRuby (Ferret), Common Lisp (Montezuma)

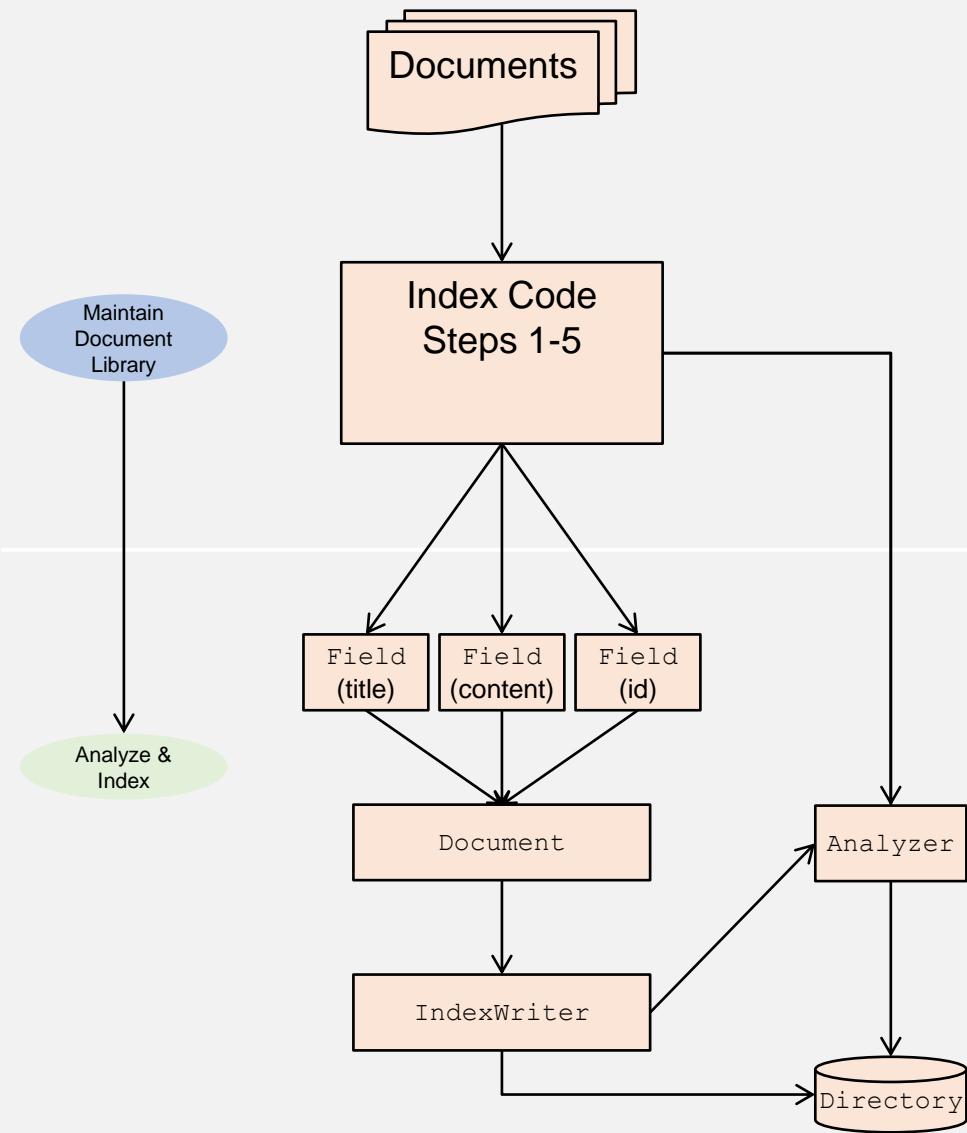
2.6.2 Core Data Model of Lucene

- Lucene is a high-performance, full-featured text search **library**. It is suitable for a wide range of applications that require text retrieval functions. Most importantly, it works across different platforms, firstly due to its Java implementation, and secondly, due to the many ports to other programming languages.
- If you are looking for an open source search engine, Lucene based projects such as Nutch (web search engine) or Solr (enterprise search engine) provide ready-to-deploy search applications. In all other cases, we have to implement the search features through the Lucene APIs.
- The core concepts of Lucene revolve around
 - Document **and** Field to encompass the content of documents
 - Analyzer to parse the content and extract features
 - IndexWriter which maintains the inverted index including concurrency control
 - Directory that holds the inverted index structures
 - Query **and** QueryParser represent queries and parse input strings, respectively
 - Term **and** TermQuery denote unit search expressions
 - IndexSearcher exposes search methods over the inverted indexes
 - TopDocs contains the result of a search sorted by scores

- Lucene's API is split into offline analysis functions and online search function. The interaction with an application is as follows:



2.6.3 Indexing Documents with Lucene



1. Select Directory to store Index in

```
directory = FSDirectory.open("./index");
```

2. Create Analyzer for Documents

```
analyzer = new StandardAnalyzer();
```

3. Create Document and add Fields

```
doc = new Document();  
doc.add(new TextField("title", title,  
        TextField.TYPE_STORED));  
doc.add(new TextField("content", content,,  
        TextField.TYPE_NOT_STORED));  
doc.add(new StoredField("id", id));
```

4. Get Index Writer and add Document

```
config = new IndexWriterConfig(analyzer);  
writer = new IndexWriter(directory, config);  
writer.addDocument(doc);
```

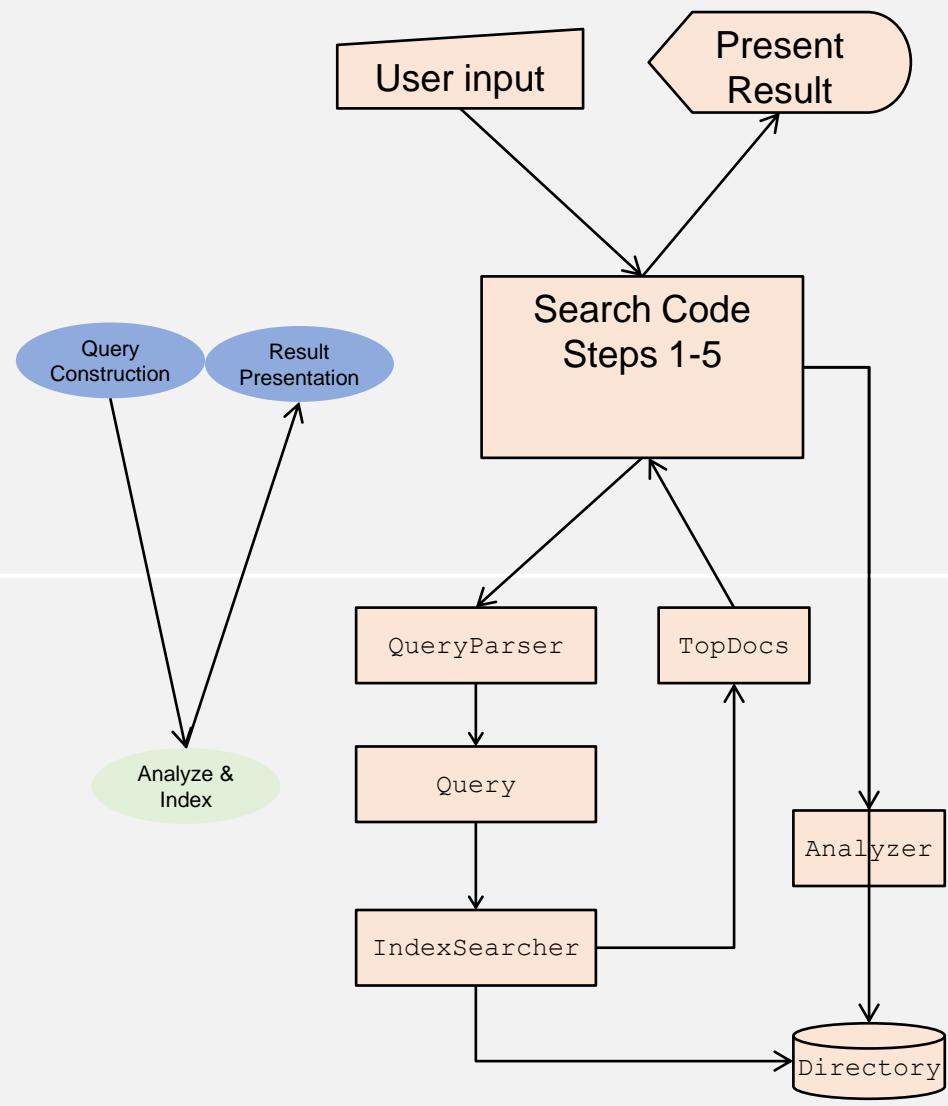
5. Close Index Writer (optionally commit / close)

```
writer.commit();  
writer.close();
```

2.6.4 Indexing Documents with Lucene

- Directory
 - Lucene provides multiple ways to maintain and persist inverted indexes. Among them are file based indexes, memory based indexes, and database indexes
 - The `LockFactory` associated with a directory implements basic concurrency control mechanisms. `IndexWriter` and `IndexSearcher` provide concurrency control to the application to ensure integrity of the indexes (other transaction attributes depend on the selected directory implementation)
- Analyzers
 - Lucene and 3rd party extensions provide a rich set of pre-defined analyzers with support for various languages. The main function of an analyzer is to return a `TokenStream`. A token stream implements a pipeline that cascades a tokenizer with a set of token filters.
 - A `Tokenizer` parses the fields of documents, removes syntactical elements, and produces a stream of tokens.
 - A `TokenFilter` filters/changes/aggregates elements in the token stream. Prominent examples include stemming, stop word elimination, and lower case converter.
- Fields
 - Lucene is able to store additional attributes for each document in the index. The purpose of fields is two-fold:
 - Ability to restrict the search on specified meta data items (e.g., only title, author, abstract, etc.)
 - Ability to store data that identify the document (or are relevant for presentational purposes)
 - Creation of fields includes many options (newer release subsumes all of them in `FieldType`)
 - `Field.Store`: YES or NO indicating whether the content needs to be stored. NO means that the content is only analyzed but not available at search time any more. Use YES for identifying attributes (or for presentation). Typical examples include ID, file name, document type, date of insertion, size of document.
 - (deprecated) `Field.Index`: main values are ANALYZED and NO. NO indicates that the field must not be analyzed; it is not possible to search for such attributes. ANALYZED is used for content that must be indexed.
 - (deprecated) `Field.TermVector`: allows to fine tune what term vector information is kept in the index.

2.6.5 Searching Documents with Lucene



1. Select Directory where Index resides

```
directory = FSDirectory.open("./index");
```

2. Create Analyzer as used for Documents

```
analyzer = new StandardAnalyzer();
```

3. Create Query (optionally through QueryParser)

```
parser = new QueryParser("content", analyzer);  
Query query = parser.parse(queryStringFromUser);
```

4. Get Index Searcher and Search

```
searcher = new IndexSearcher(directory);  
TopDocs hits = searcher.search(query, NUM_RESULTS);
```

5. Present Result

```
for(int i=0;i<hits.scoreDocs.length;i++){  
    doc = searcher.doc(hits.scoreDocs[i].doc);  
    System.out.printf(" %4d %1.3f %s %s\n",  
        i+1,  
        hits.scoreDocs[i].score,  
        doc.get("id"), doc.get("title"));  
}
```

2.6.6 Searching Documents with Lucene

- Query and QueryParser
 - Lucene provides multiple ways to query the content of an index. Queries are always against the content of analyzed field data. Atomic queries consist of term queries, range queries, phrase queries, fuzzy queries (searching for all terms that are close to the given one), wildcard queries and so on. Atomic queries can be combined by means of Boolean operators.
 - QueryParser simplifies the interface with a standard way how users have to enter queries
 - Query is a set of clauses optionally prefixed with '+' (must include) and '-' (must not be included)
 - A clause can be a single term such as 'hello' for the default search field, a search term for a selected field such as 'title:hello', a fuzzy query such as 'hello~' or-ing all similar terms in the index, a wildcard-query such as 'h?llo' or-ing all matching terms, and many more.
 - Scores are computed through a Similarity object. The example code uses the default scoring, but it is possible to overwrite how Lucene scores and ranks documents (see next slide)
- TopDocs
 - The search method of the IndexSearcher returns the top (NUM_RESULTS) documents matching the query and ordered by their score.
 - Retrieval of the content of fields of document is through the IndexSearcher. TopDocs only holds Lucene internal document identifiers (property doc of scoreDocs field in TopDocs).
 - Only fields that were indexed with Field.Store.YES can be retrieved after a search. Any other metadata has to be retrieved by the application it self.
- Analyzer
 - Use the same analyzer object as for indexing the documents in offline mode. Lucene provides versioned standard analyzer to avoid confusion should the standard implementation change over time.

2.6.7 Retrieval Model of Lucene

- Lucene combines Boolean retrieval with vector space retrieval. Only documents that match the Boolean query are considered. These candidates are scored with an extended version of the vector space retrieval model, and the top k documents are returned.
- Boolean Retrieval Part
 - Applications can define arbitrary Boolean expressions on fields content with
 - atomic queries such `TermQuery`, `RangeQuery`, or any other `Query`
 - and a Boolean clause constraint whether `MUST`, `MUST NOT`, or `SHOULD` occur
 - Example: `+information -multimedia` retrieval search

```
TermQuery q1 = new TermQuery(new Term("content", "information"));
TermQuery q2 = new TermQuery(new Term("content", "retrieval"));
TermQuery q3 = new TermQuery(new Term("content", "search"));
TermQuery q4 = new TermQuery(new Term("content", "multimedia"));
BooleanQuery.Builder builder = new BooleanQuery.Builder();
builder.add(q1, BooleanClause.Occur.MUST);
builder.add(q2, BooleanClause.Occur.SHOULD);
builder.add(q3, BooleanClause.Occur.SHOULD);
builder.add(q4, BooleanClause.Occur.MUST_NOT);
BooleanQuery query=builder.build();
```

- `FuzzyQuery` and `WildcardQuery` translate into a `MultiTermQuery` over a set of terms
 - `FuzzyQuery` ('hello~0.5') expands to a search over all terms in the index that have a normalized similarity of 0.5 and larger (value btw 0 and 1). Similarity is measured with edit distance and normalized over the length of the term.
 - `WildcardQuery` ('h?llo') expands to a search over all terms that match the pattern

- Boolean Retrieval Part (contd)
 - IndexSearcher uses the inverted lists in the directory to retrieve all documents that match the Boolean condition. This is the set of candidates.
- Ranking uses an extended version of the cosine measure. However, there are several additional factors and normalizations built into the standard similarity measure
 - The conceptual scoring formula is:

$$\text{score}(q, d) = \text{coord_factor}(q, d) \cdot \text{query_boost}(q) \cdot \frac{V(q) \cdot V(d)}{\|V(q)\|} \cdot \text{doc_len_norm}(d)$$

- $\text{coord_factor}(q, d)$: score factor based on how many query terms are found in the document. In essence, this scores how many of the optional terms (OR clauses) are found in d .
- $\text{query_boost}(q)$: boost factor for individual query terms to be taken into account
- $V(q), V(d)$: vector representation i.e., tf*idf weighted number of term occurrences
- $\text{doc_len_norm}(d)$: unlike the normalization of queries by their length, documents are normalized by the length of a field (number of term occurrences) to boost smaller fields over larger fields
- To simplify computation, Lucene's implementation is as follows
 - query norm and query boost are combined as they are known at search start time
 - document norm and document (field) boost values are stored in the index for each term

- Ranking in Lucene (contd)
 - The formula defined by `TFIDFSimilarity` is:

$$score(q, d) = coord(d, q) \cdot queryNorm(q) \cdot \sum_{t \text{ in } q} (tf(t, d) \cdot idf(t)^2 \cdot boost(t) \cdot norm(t, d))$$

- $coord(d, q) = \frac{overlap}{max_overlap}$ boosts documents that contain more of the query terms (not the number of occurrences. `max_overlap` is the maximum number of query terms found in a single document.
- $queryNorm(q) = \frac{1}{(boost(q)^2 \cdot \sum_{t \text{ in } q} (idf(t) \cdot boost(t))^2)}$ is used to make scores of different (sub-)queries comparable. It does not affect document ranking (constant factor) but how a query overall is weighted. `boost(q)` is an application specified boost factor for the query.
- $tf(t, d) = \sqrt{frequency \text{ of } t \text{ in } d}$ documents with higher numbers of term occurrences obtain a higher weight. Note the query term occurrences are not taken into account. Rather, Lucene treats each term occurrence the same, e.g., if the term occurs twice, two sub-queries exist for weighting
- $idf(t) = 1 + \log \left(\frac{numDocs}{docFrequency+1} \right)$ denotes the standard inverse document frequency applied to both query and document terms
- $boost(t)$ represents an application specified boost value for a term t in the query
- $norm(t, d) = boost(d) \cdot \frac{1}{number \text{ of } terms \text{ in } field} \cdot \prod_{\text{field } f \text{ in } d \text{ anmeds as } t} boost(f)$ denotes a value that Lucene computes at indexing time and stores within the inverted lists for each term in document d . `boost(d)` denotes a boost factor that applications can specify when adding documents.

2.7 Literature and Links

General Books on Text Retrieval

- Gerard Salton and Michael J. McGill. Information Retrieval - Grundlegendes für Informationswissenschaftler, McGraw-Hill Book Company, 1983.
- W.B. Frakes and R. Baeza-Yates. Information Retrieval, Data Structures and Algorithms, Prentice Hall, 1992.
- Karen Sparck Jones and Peter Willet. Readings in Information Retrieval. Morgan Kaufmann Publishers Inc., 1997.
- David A. Grossmann and Ophir Frieder. Information Retrieval: Algorithms and Heuristics, Kluwer Academic Publishers, 1998 (1st edition), 2004 (2nd edition).
- Ricardo Baeza-Yates and Berthier Ribeiro-Neto. Modern Information Retrieval, ACM Press Books, 1999 (1st edition), 2011 (2nd edition).
- Sandor Dominich. Mathematical Foundations of Information Retrieval, Kluwer Academic Publishers, 2001.
- Christopher Manning, Prabhakar Raghavan, Hinrich Schütze. Introduction to Information Retrieval, Cambridge University Press, 2008
- Stefan Büttcher, Charles Clarke, Gordon Cormack. Information Retrieval - Implementing and Evaluating Search Engines. MIT Press 2010.

Implementations

- Apache Lucene, <https://lucene.apache.org>