Universität Basel

Computer Science / 15731-01 / 2023

Multimedia Retrieval

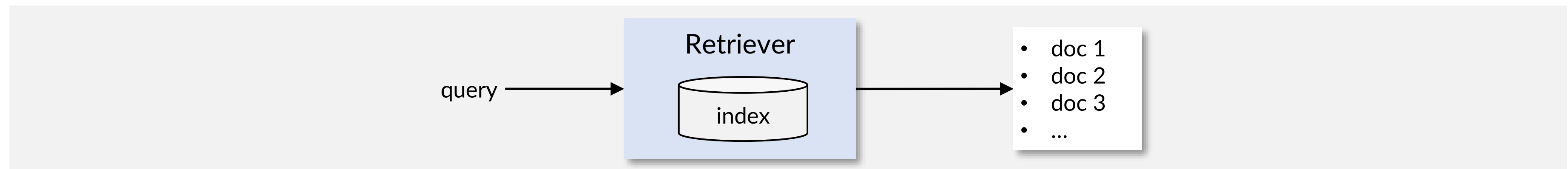
Chapter 3: Classical Text Retrieval

3.1 Introduction
3.2 Fundamentals
3.3 Text Retrieval Models
3.4 Indexing Structures
3.5 Lucene - Open Source Text Search
3.6 Literature and Links

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

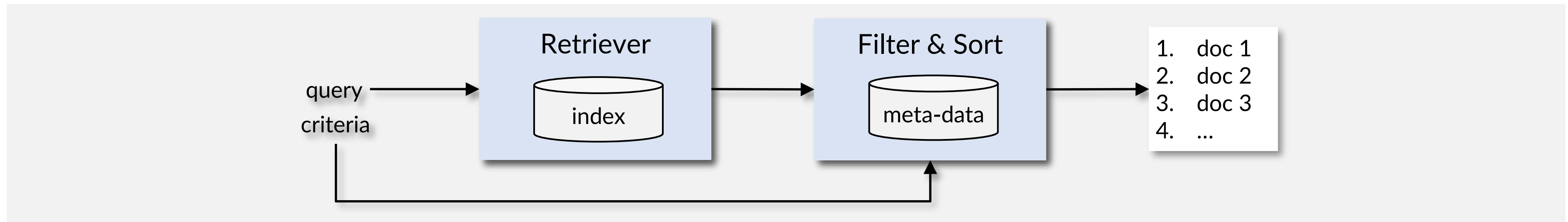
3.1 Introduction

- Text retrieval originated in the 1950s and 1960s through pioneering research by Gerard Salton, Karen Spärck Jones, and others. It became popular due to its wide range of applications, simplicity, and user-friendly interface. As discussed earlier, text retrieval is less affected by the semantic gap compared to other media types (although this will be further discussed in upcoming chapters). Users input text queries against unstructured documents, and the systems can easily match the query with the document, as they share the same representation. Additionally, textual metadata enables any media type to be searchable using the same approach.
- This allowed the relatively basic computer systems back then to offer efficient and effective search for expert users. As early computers had limitations in terms of storage and compute, models progressed from simple Boolean matching to more complex vector space and probabilistic models as technology improved. The first generation primarily focused on "Retriever-only" models.



  - **Boolean Retrieval Systems** hold a significant advantage as they can determine document relevance while scanning the data, without the need for post-processing to sort and rank documents. Additional filters, such as publication date or author, can be easily integrated into the Boolean model. This builds a robust foundation still observed in today's systems like when searching for files on a local drive
  - The Boolean Model uses set theory and Boolean algebra. Documents are represented as a set of terms, without considering the number of occurrences. The query is formulated as a Boolean expression using operators like AND and OR to combine term match atomic queries. If a document satisfies the Boolean expression (and other filter conditions on its metadata), it is included in the result set; otherwise, it is excluded
  - Boolean models do not use scoring or ranking, so they can return results as soon as they find the first matching document while scanning the data (consider the example of to searching through a local hard drive). In addition, they can utilize a simple index structure called **inverted file** which makes the search process very efficient by considering only a small fraction of the data. This method is still used in modern algorithms today.

- As collections grew larger, the Boolean model needed an extension for better result organization and exploration. When there are hundreds of hits, users want a more efficient way to browse through search results. A post-processing step was introduced, enabling query-independent filtering and sorting, such as sorting by publication date or filtering for a specific language. Unlike the retriever step, users can add or remove filters and change sorting while exploring the results and without re-submitting the search. In other words, this post-processing does not impact the set of relevant documents and is often implemented in the interface directly:

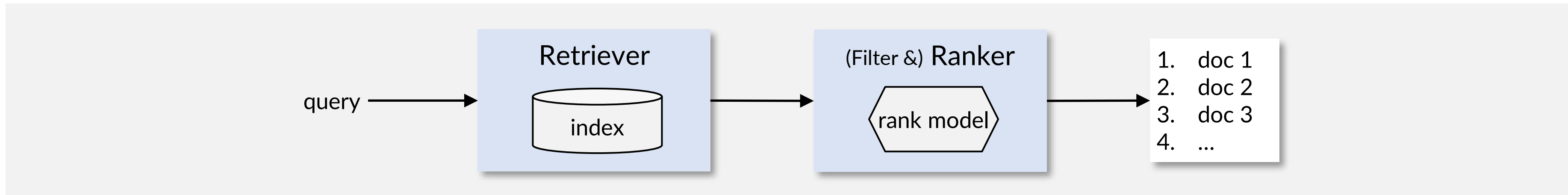


- The above method works well for scenarios where exploration is mostly focused on metadata, as in shop or library searches. However, a key drawback is that sorting does not consider how well an object fits the query. Early extensions of the Boolean model (**Extended Boolean Model**) addressed this limitation by studying the impact of the query terms' presence in documents and their relevance assessment. For example, consider the query "cat AND dog" and the three documents:
    1) "A cat walked down the street."
    2) "The dog chased the cat."
    3) "The cat played with the dog when another cat and dog approached them."

  Documents 2) and 3) meet the condition "cat AND dog“, but document 1) is dismissed by the Boolean logic although it appears partially relevant to the query. Furthermore, document 3) contains the query terms more frequently and seems to be a better fit for the query, but the Boolean expression classifies documents 2) and 3) the same. The **Extended Boolean Model**, changes the foundation model in two ways:
  – It allows for partial matches to the query (like Document 1) but assigns them lower relevance scores
  – It considers how often query terms appear in documents when calculating relevance scores

  Using these relevance scores, we can sort the document collection and present results even if not all conditions are met. In other words, instead of using "hard" conditions, we apply penalties for not meeting the condition.

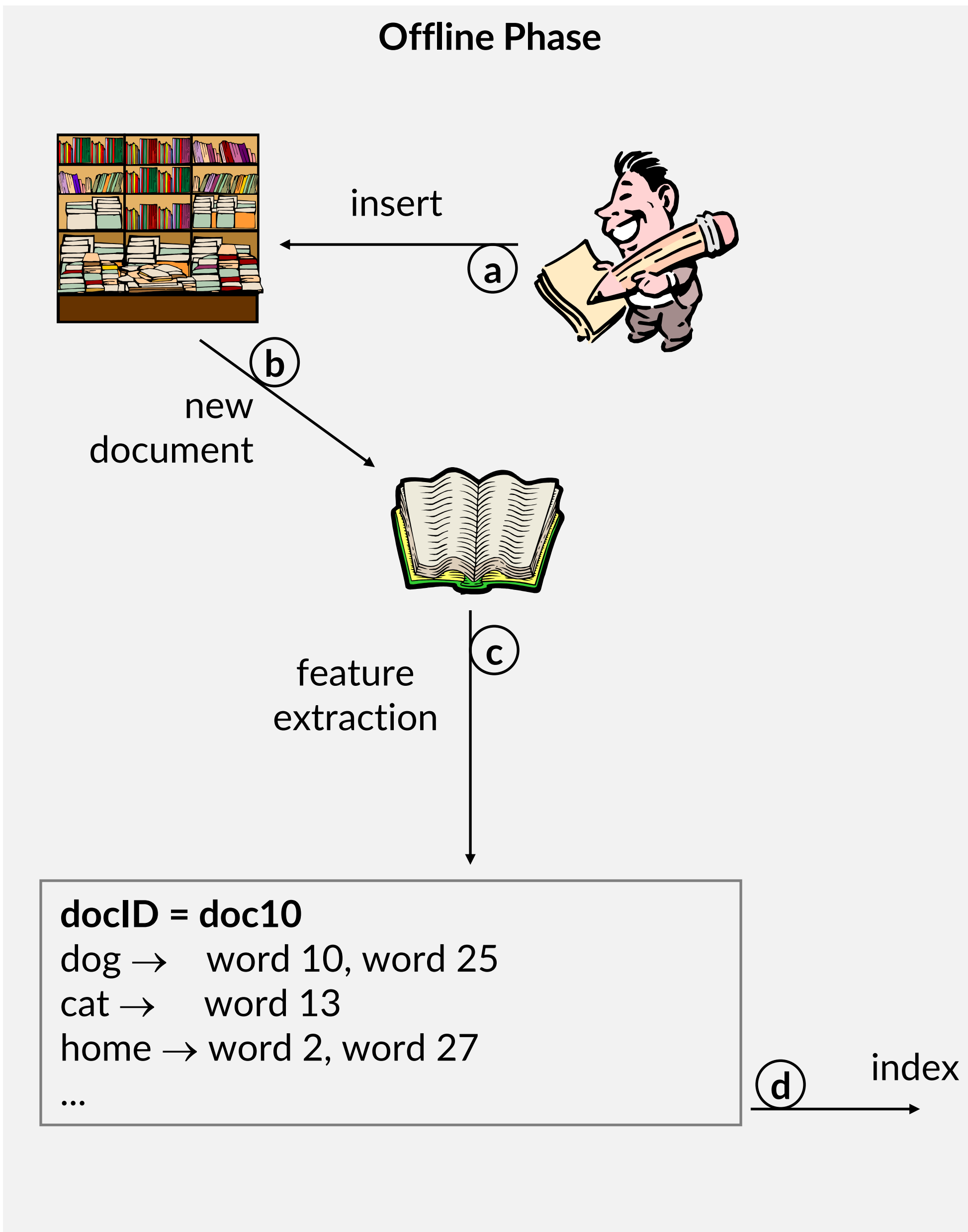
- In the 1970s, classical **Vector Space** and **Probabilistic Retrieval** models emerged. Both methods established a relevance model for document-query matching. In the vector space model, documents and queries are represented as high-dimensional vectors, and heuristic methods compare vectors to obtain a notion for relevance. Probabilistic retrieval models assume that documents are generated randomly from a probabilistic model, and relevance is determined by the probability of a document being relevant to the query. Newer models like **BM25** combine vector space and probabilistic retrieval techniques.



  Extended Boolean Model, Vector Space Retrieval, and Probabilistic Retrieval follow a similar approach: a retriever gathers a larger set of candidate documents based on query terms, and a ranking model assesses the relevance to produce a sorted result list. Further filter conditions can be applied to explore the result collection, such as language filtering or year of publication.

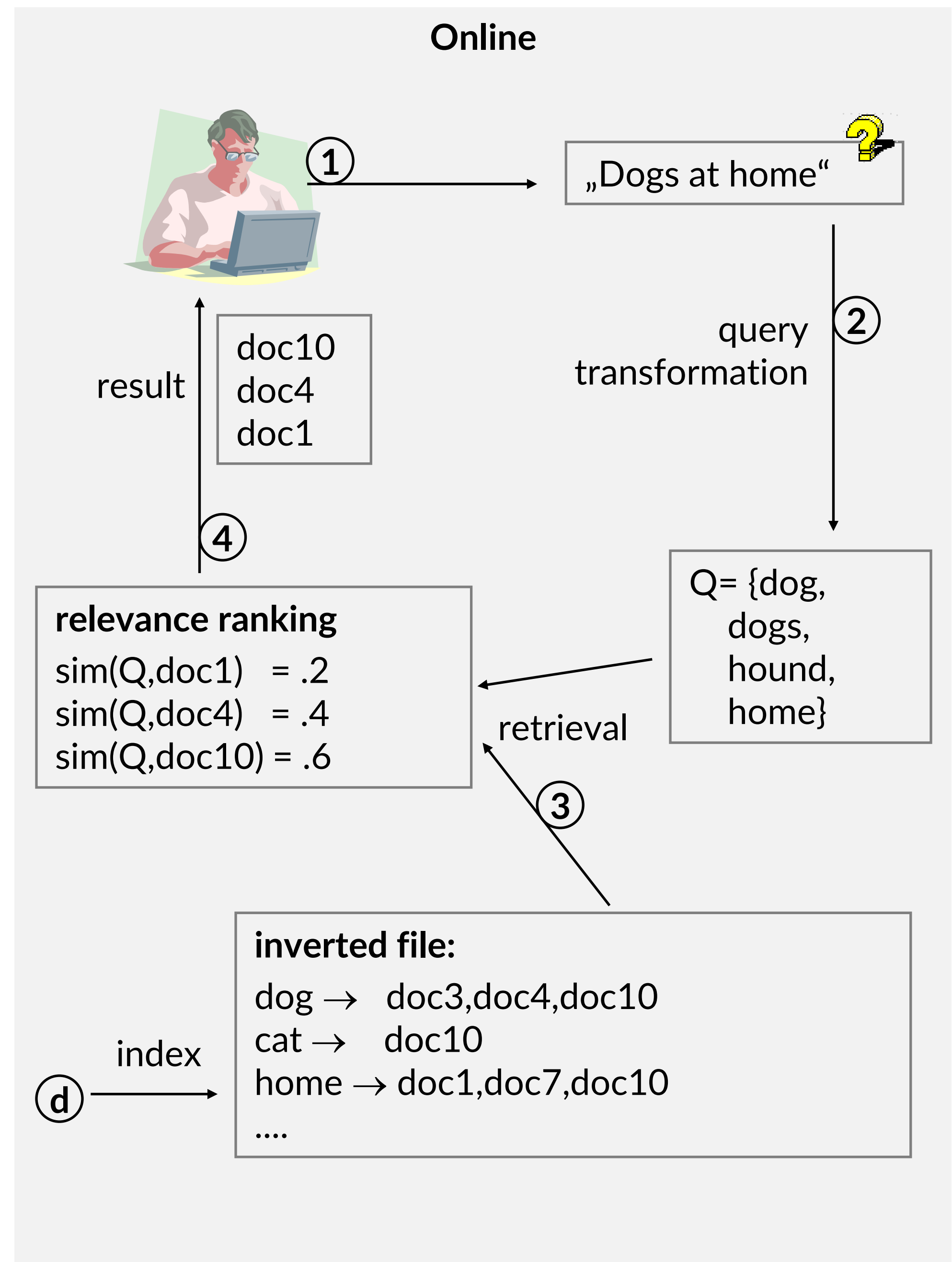
- In this chapter, we delve into classical text retrieval models in detail:
  - We begin by exploring document descriptions and performing simple linguistic operations to reduce words to terms, forming a vocabulary for search
  - Next, we study classical models like the Standard and Extended Boolean Model, Vector Space Retrieval Model, Probabilistic Model, and the modern BM25 model used in popular software packages
  - We then examine indexing methods, notably inverted file, and a simple implementation using a relational database to accelerate the search process
  - Finally, we conclude the chapter by discussing Apache Lucene, a popular software packages that offer state-of-the-art text retrieval for various platforms
- In the chapters to follow this one, we will explore: 1) natural language processing and advanced techniques for generating vectors from text representations, 2) web retrieval as a unique search challenge, and 3) modern AI-supported classification and search methods.

3.2 Fundamentals

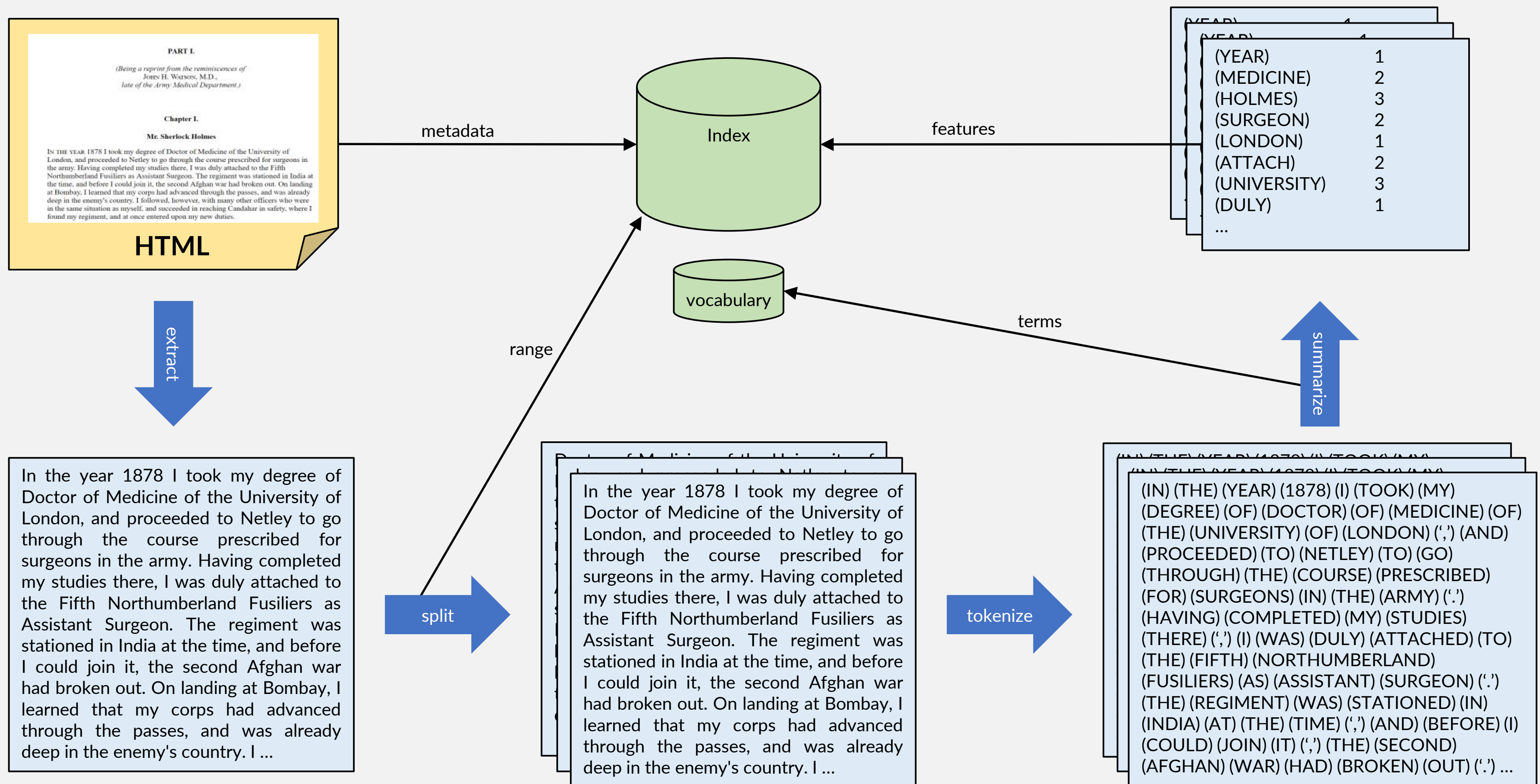


- Many search systems, like searching through files on a local drive, scan through all the data for each query. However, this approach is not efficient for large text collections. Instead, the search is divided into two parts: an offline indexing phase (depicted on the left) and an online querying phase (see next page).
- The **offline phase** extracts meaningful features from text documents and stores them, along with metadata, in an index for future query use. These features provide a concise representation of the document's content and are typically represented by high-dimensional vectors.
- During the offline mode, the following steps take place:
  a) add a new document (or find one by scanning/crawling)
  b) each addition triggers feature extraction and updates search indexes
  c) extract features that best describe the content, analyze context, and include higher-level features
  d) pass the features to an index that accelerates searches for queries
- The main challenge lies in extracting concise representations from the documents. In this chapter, we will use simple methods to create vector representations. In the chapters to follow this one, we will explore more advanced techniques.

- In the online mode, users can search for documents using the indexed data from the offline phase. The query is analyzed similarly to the documents with additional processing to correct spelling mistakes or include synonyms for a broader search. Retrieval involves comparing features. If two documents have similar features, they are considered similar in content. Thus, a document is considered a good match to a query if its features are close to those of the query.
- In the online mode, the following steps take place:
  1) user enters a query (or speech/handwriting recognition)
  2) we extract features from the query, similar to the process for documents, and transform the query as needed (e.g., correcting spelling mistakes)
  3) we use the query features to search the index for documents with similar features
  4) we rank the documents based on their retrieval status value (RSV) and return the best-matching documents
- The primary challenge is relevance ranking. The goal is to accurately assess a document's relevance based solely on its feature representation, and given the features of the query. In subsequent chapters, we will explore more sophisticated methods, including generative AI. However, in this chapter, we will use simple yet efficient and effective methods that are suitable for many use cases.



- In the rest of this chapter, we explore the fundamental steps to extract features from source documents ("offline phase"), as mentioned earlier. The overall process is detailed in the picture below. We will discuss indexing in a later section, focusing here on four fundamental actions during feature extraction: 1) extract, 2) split, 3) tokenize, and 4) summarize. The outcome includes a vocabulary containing all terms found in the documents, which is also used for query analysis. Additionally, we obtain for each document chunk out of the splitting step a feature representation that we can store in an index along with metadata from the source document and split ranges (start and end coordinates in the source document).



3.2.1 Action 1: Extract (with the example of HTML)

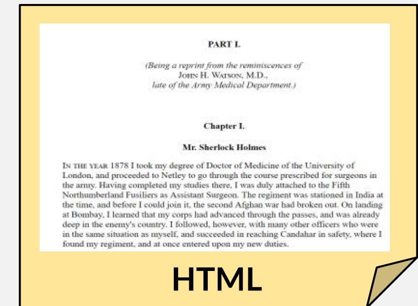
- Text documents are available in different formats such as HTML, PDF, EPUB, metadata, or plain text. The first step involves extracting meta information and the sequence of characters that form the text stream without control sequences and formatting information present in the source document. This may include structural analysis of the document, encoding adjustments, and identifying relevant information for feature extraction. In some cases, we may have to apply text extraction from images.
- Consider a simple example in HTML with the following snippet representing a web page's structure. The initial task is to identify the useful bits of information within it. The header typically holds rich meta information, while the body contains the main text parts. Although HTML follows a well-defined standard, extracting information (known as scraping) requires analyzing the data structure used for the pages. In contrast, a web search engine considers everything present on the page.

```
<html>
    <head>
        <title> MMIR – 2023 </title>
        <meta name="keywords"
              content="multimedia, retrieval, course"/>
    </head>

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

Body: Contains the main content enriched with markups. The document's flow is not always obvious and may appear differently on screen than in the file.

Header: Contains meta-information about the document. We can utilize this information to add relevant metadata for the document (and its chunks).

- At this point, we must decide for a character encoding that we will use for the terms and the index, and convert the source text. UTF-8/16/32 are widely used but can limit the ability to support different languages.



PART I.

(Being a reprint from the reminiscences of
JOHN H. WATSON, M.D.,
late of the Army Medical Department.)

Chapter I.

Mr. Sherlock Holmes

IN THE YEAR 1878 I took my degree of Doctor of Medicine of the University of London, and proceeded to Netley to go through the course prescribed for surgeons in the army. Having completed my studies there, I was duly attached to the Fifth Northumberland Fusiliers as Assistant Surgeon. The regiment was stationed in India at the time, and before I could join it, the second Afghan war had broken out. On landing at Bombay, I learned that my corps had advanced through the passes, and was already deep in the enemy's country. I followed, however, with many other officers who were in the same situation as myself, and succeeded in reaching Candahar in safety, where I found my regiment, and at once entered upon my new duties.

HTML

In the year 1878 I took my degree of Doctor of Medicine of the University of London, and proceeded to Netley to go through the course prescribed for surgeons in the army. Having completed my studies there, I was duly attached to the Fifth Northumberland Fusiliers as Assistant Surgeon. The regiment was stationed in India at the time, and before I could join it, the second Afghan war had broken out. On landing at Bombay, I learned that my corps had advanced through the passes, and was already deep in the enemy's country. I …

- Let's use the example of HTML to illustrate some aspects for metadata generation:
  - **URI of page**: both metadata and content (may serve concise key words for retrieval)

    ```
    https://dmi.unibas.ch/de/studium/computer-science-informatik/lehrangebot-hs23/lecture-multimedia-retrieval/
    ```

  - **Title of document**: both metadata and content (may serve concise key words for retrieval)

    ```html
    <title>Multimedia Retrieval - Homepage</title>
    ```

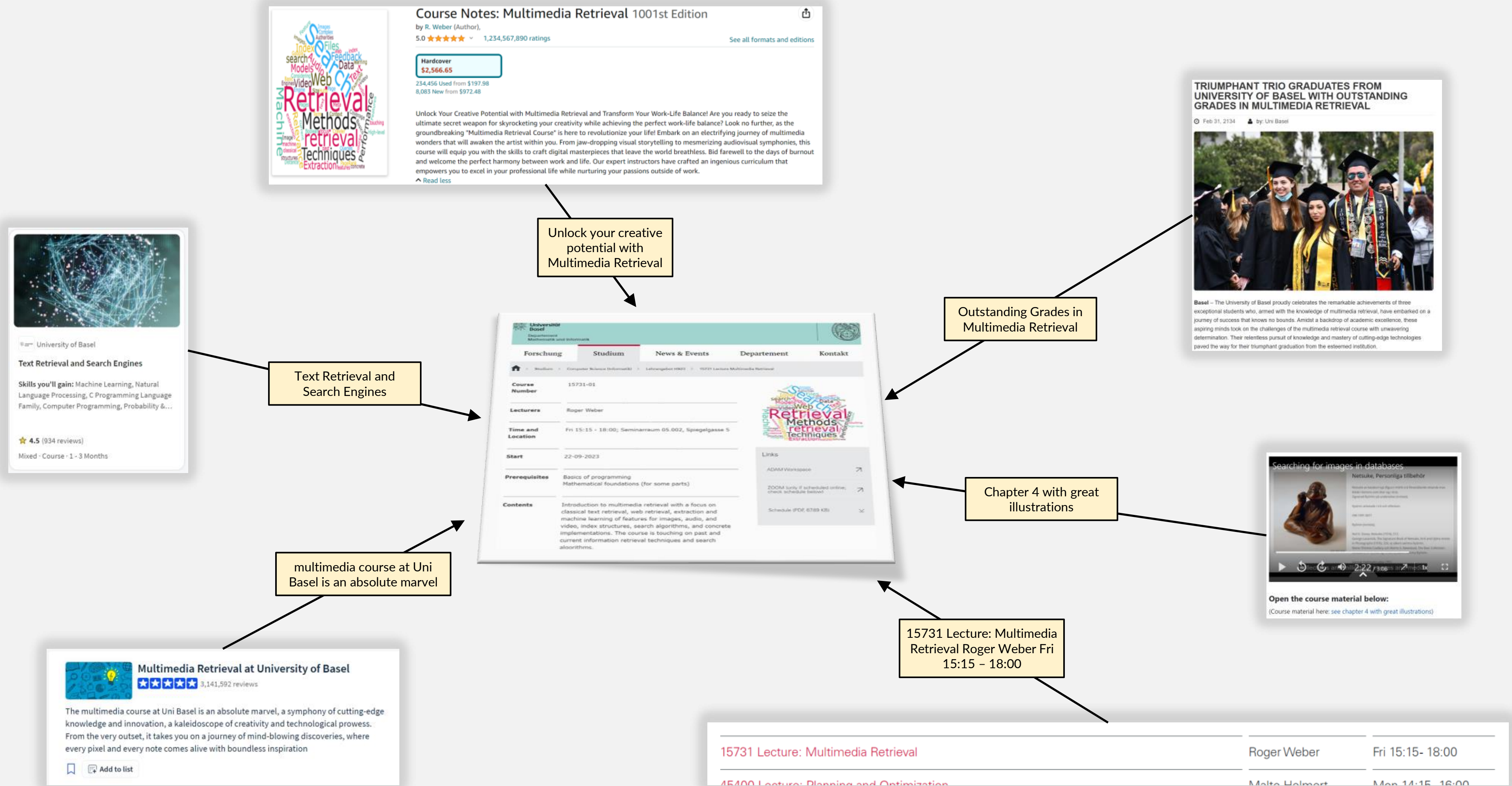
  - **Meta information** in header section: (enriched information provided by author)

    ```html
    <meta name="keywords" content="MMIR, information, retrieval">
    <meta name="description" content="This will change your life…">
    ```

  - As we discussed in the metadata section, we must be cautious about its reliability. It might include false information or describe aspects differently from what we observed in other documents. Nevertheless, in many cases, the brief nature of metadata allows us to assign high weights to the text parts.
- Web pages contain links. How do we handle them effectively? Links describe relationships between documents and can enhance the current document's description. More importantly, they also describe the referenced document. Since web page authors often use concise anchor texts, the keywords in anchor texts serve as an excellent source of additional terms for the referenced document. Usually, the link text is associated with both the embedding and linked documents. However, we typically give much higher weight to keywords for the referenced document. It is essential to consider the approach's effectiveness, especially when dealing with click baits (promising more than the referenced documents reveal) or navigational hints like "click here" or "back to the main page". These keywords add no additional content for the referenced document.
- The body includes all text blocks and uses tags to control rendering. The page's flow may not exactly match the order in the HTML file, but it's usually a good enough approximation. Certain tags offer valuable additional information on the following text pieces. For example, we can assign higher weights to term occurrences in headlines, bold text, or text with emphasized rendering on the page.
- HTML includes escape sequences for special characters that need to be translated into the target encoding format.

  ```
      ->  space        &uuml;   ->   ü
  ```

- The illustration below shows how anchor texts (and their surroundings) provide relevant terms for describing target pages (and images). We emphasized the need for caution with human metadata. However, anchor texts come from diverse sources, simplifying the identification of useful terms across all mentions and filtering out "outliers" with obviously incorrect information. In a subsequent chapter, we will delve into using the link network to assess a page's importance and (objective) relevance through PageRank.



3.2.2 Action 2: Split

- Most traditional retrieval methods are optimized for smaller documents. This is because they assign a single term vector to represent the entire document. For instance, a 3-page document and a 1000-page novel are both described using a single vector.
  – In cases where the document is small, returning the entire document to users is acceptable as they can easily find the relevant location within it. However, with larger documents like novels, it becomes essential to provide additional information on the specific passage's location. Splitting the documents into smaller pieces allows for a more precise retrieval at the expense of having more data entries in the collection.
  – Another reason is that many traditional retrieval models do not include support for proximity metrics in their relevance assessment. For example, a query like "cats AND dogs" could retrieve a novel containing the term "cats" only on the first page and "dogs" only on the last page. Splitting documents into smaller chunks enforces proximity between query terms. For instance, if we split the novel by chapter, the novel and its chapters are no longer relevant for the query as none of the chapters contain both "cats" and "dogs".
- There is no one-size-fits all solution for splitting documents. In general, it is a trade-off between more and smaller but semantically coherent parts of the documents, and additional costs for storage and retrieval:
  – For instance, splitting a novel by sentences may create too many entries that negatively impacts performance given a library with thousands of books. Sentences may also be too narrow for finding meaningful matches for more complex queries
  – On the other side, a search engine for citations in religious texts may split documents at the sentence or verse level to create thousands of smaller parts that can be individually retrieved with searches

In the year 1878 I took my degree of Doctor of Medicine of the University of London, and proceeded to Netley to go through the course prescribed for surgeons in the army. Having completed my studies there, I was duly attached to the Fifth Northumberland Fusiliers as Assistant Surgeon. The regiment was stationed in India at the time, and before I could join it, the second Afghan war had broken out. On landing at Bombay, I learned that my corps had advanced through the passes, and was already deep in the enemy's country. I ...

In the year 1878 I took my degree of Doctor of Medicine of the University of London, and proceeded to Netley to go through the course prescribed for surgeons in the army. Having completed my studies there, I was duly attached to the Fifth Northumberland Fusiliers as Assistant Surgeon. The regiment was stationed in India at the time, and before I could join it, the second Afghan war had broken out. On landing at Bombay, I learned that my corps had advanced through the passes, and was already deep in the enemy's country. I ...

- **Method 1:** Splitting the text into fixed-sized chunks
  - The document is divided into chunks with a constant number of tokens, such as words or characters. This approach is straightforward and has even sizes for all document chunks simplifying normalization
  - At the chunk boundaries, we may encounter half-sentences and splits of passages that belong together such as a paragraph in a document
  - In the example on the right, the chunks are split after every 50 tokens. The number of tokens used for splitting is a hyperparameter and requires training to achieve optimal results in the given search context

> - In the year 1878 I took my degree of Doctor of Medicine of the University of London, and proceeded to Netley to go through the course prescribed for surgeons in the army. Having completed my studies there, I was duly attached to the Fifth Northumberland Fusiliers as
> - Assistant Surgeon. The regiment was stationed in India at the time, and before I could join it, the second Afghan war had broken out. On landing at Bombay, I learned that my corps had advanced through the passes, and was already deep in the
> - enemy's country. I followed, however, with many other officers who were in the same situation as myself, and succeeded in reaching Candahar in safety, where I found my regiment, and at once entered upon my new duties. The campaign brought honours and promotion
> - to many, but for me it had nothing but misfortune and disaster. I was removed from my brigade and attached to the Berkshires, with whom I served at the fatal battle of Maiwand. There I was struck on the shoulder by a Jezail bullet, which
> - shattered the bone and grazed the subclavian artery. I should have fallen into the hands of the murderous Ghazis had it not been for the devotion and courage shown by Murray, my orderly, who threw me across a pack-horse, and succeeded in bringing me safely to
> - the British lines. Worn with pain, and weak from the prolonged hardships which I had undergone, I was removed, with a great train of wounded sufferers, to the base hospital at Peshawar. Here I rallied, and had already improved so far as to
> - …

- **Method 2:** Splitting the text with NLP techniques
  - Using NLP methods, the text is first divided into sentences. Then, several sentences are combined until a minimum number of tokens is reached
  - At chunk boundaries, we no longer observe half-sentences (unless sentence segmentation was incorrect), but we might still split passages that belong together, such as a paragraph in a document
  - In the example on the right, we first split the text into sentences and then combine them until each chunk contains at least 50 tokens. As mentioned before, the number of tokens used for splitting is a hyperparameter that requires training to achieve optimal results in the given search context.
  - Chunks sizes can now vary in length and variations depend on the length of sentences.

> - In the year 1878 I took my degree of Doctor of Medicine of the University of London, and proceeded to Netley to go through the course prescribed for surgeons in the army. Having completed my studies there, I was duly attached to the Fifth Northumberland Fusiliers as Assistant Surgeon.
> - The regiment was stationed in India at the time, and before I could join it, the second Afghan war had broken out. On landing at Bombay, I learned that my corps had advanced through the passes, and was already deep in the enemy's country.
> - I followed, however, with many other officers who were in the same situation as myself, and succeeded in reaching Candahar in safety, where I found my regiment, and at once entered upon my new duties. The campaign brought honours and promotion to many, but for me it had nothing but misfortune and disaster.
> - I was removed from my brigade and attached to the Berkshires, with whom I served at the fatal battle of Maiwand. There I was struck on the shoulder by a Jezail bullet, which shattered the bone and grazed the subclavian artery. I should have fallen into the hands of the murderous Ghazis had it not been for the devotion and courage shown by Murray, my orderly, who threw me across a pack-horse, and succeeded in bringing me safely to the British lines.
> - Worn with pain, and weak from the prolonged hardships which I had undergone, I was removed, with a great train of wounded sufferers, to the base hospital at Peshawar. Here I rallied, and had already improved so far as to be able to walk about the wards, and even to bask a little upon the verandah, when I was struck down by enteric fever, that curse of our Indian possessions.
> - For months my life was despaired of, and when at last I came to myself and became convalescent, I was so weak and emaciated that a medical board determined that not a day should be lost in sending me back to England. I was dispatched, accordingly, in the troopship "Orontes," and landed a month later on Portsmouth jetty, with my health irretrievably ruined, but with permission from a paternal government to spend the next nine months in attempting to improve it.
> - …

- **Method 3**: Metadata or structural information
  - If the document contains metadata or structural markers for paragraphs, sections, chapters, or pages, we can use these markers as chunk boundaries. With plain text, we can also look for paragraphs often marked with a newline character or an empty line
  - Chunks now are contextually coherent like a full paragraph in a document. But we have considerable differences in the number of tokens per chunk that require normalization during the ranking process (see BM25 later for an example)
  - In the example on the right, we split at the end of a paragraph. Especially in novels with spoken sentences, it sometimes is not so obvious where a paragraph ends

> - In the year 1878 I took my degree of Doctor of Medicine of the University of London, and proceeded to Netley to go through the course prescribed for surgeons in the army. Having completed my studies there, I was duly attached to the Fifth Northumberland Fusiliers as Assistant Surgeon. The regiment was stationed in India at the time, and before I could join it, the second Afghan war had broken out. On landing at Bombay, I learned that my corps had advanced through the passes, and was already deep in the enemy's country. I followed, however, with many other officers who were in the same situation as myself, and succeeded in reaching Candahar in safety, where I found my regiment, and at once entered upon my new duties.
> - The campaign brought honours and promotion to many, but for me it had nothing but misfortune and disaster. I was removed from my brigade and attached to the Berkshires, with whom I served at the fatal battle of Maiwand. There I was struck on the shoulder by a Jezail bullet, which shattered the bone and grazed the subclavian artery. I should have fallen into the hands of the murderous Ghazis had it not been for the devotion and courage shown by Murray, my orderly, who threw me across a pack-horse, and succeeded in bringing me safely to the British lines.
> - Worn with pain, and weak from the prolonged hardships which I had undergone, I was removed, with a great train of wounded sufferers, to the base hospital at Peshawar. Here I rallied, and had already improved so far as to be able to walk about the wards, and even to bask a little upon the verandah, when I was struck down by enteric fever, that curse of our Indian possessions. For months my life was despaired of, and when at last I came to myself and became convalescent, I was so weak and emaciated that a medical board determined that not a day should be lost in sending me back to England. I was dispatched, accordingly, in the troopship "Orontes," and landed a month later on Portsmouth jetty, with my health irretrievably ruined, but with permission from a paternal government to spend the next nine months in attempting to improve it.
> - …

- **Method 4:** Semantic splitting
  - The text is initially divided into smaller parts, such as sentences. By using machine learning techniques, sentences with similar topics and concepts are grouped or clustered together
  - Chunks are contextually coherent and may encompass multiple passages and paragraphs from the source document. But it may also split paragraphs or sections if topics change
  - As mentioned before, we encounter chunks with significantly different numbers of tokens
  - In the example on the right, we merged sentences that semantically belong together

> - In the year 1878 I took my degree of Doctor of Medicine of the University of London, and proceeded to Netley to go through the course prescribed for surgeons in the army. Having completed my studies there, I was duly attached to the Fifth Northumberland Fusiliers as Assistant Surgeon.
> - The regiment was stationed in India at the time, and before I could join it, the second Afghan war had broken out. On landing at Bombay, I learned that my corps had advanced through the passes, and was already deep in the enemy's country. I followed, however, with many other officers who were in the same situation as myself, and succeeded in reaching Candahar in safety, where I found my regiment, and at once entered upon my new duties. The campaign brought honours and promotion to many, but for me it had nothing but misfortune and disaster. I was removed from my brigade and attached to the Berkshires, with whom I served at the fatal battle of Maiwand. There I was struck on the shoulder by a Jezail bullet, which shattered the bone and grazed the subclavian artery. I should have fallen into the hands of the murderous Ghazis had it not been for the devotion and courage shown by Murray, my orderly, who threw me across a pack-horse, and succeeded in bringing me safely to the British lines. Worn with pain, and weak from the prolonged hardships which I had undergone, I was removed, with a great train of wounded sufferers, to the base hospital at Peshawar.
> - Here I rallied, and had already improved so far as to be able to walk about the wards, and even to bask a little upon the verandah, when I was struck down by enteric fever, that curse of our Indian possessions. For months my life was despaired of, and when at last I came to myself and became convalescent, I was so weak and emaciated that a medical board determined that not a day should be lost in sending me back to England. I was dispatched, accordingly, in the troopship "Orontes," …
> - …

3.2.3 Action 3: Tokenize

- A token is formed by a sequence of characters. Typically, we use complete words to create tokens, but there are other options which we will explore later in this course. Here's a brief overview:
  - **Characters and fragments of words** can be used to form tokens. For example, breaking the character stream into tokens of 3 characters would turn "street" into "str" and "eet". This method is frequently employed by large language models to maintain a small and constant-sized vocabulary while still being able to encode previously unseen words
  - **Words** are the primary approach used in classical text retrieval. However, we require additional definitions for special characters, numbers, and abbreviations. In certain languages, word boundaries may not always be evident (e.g., Japanese and Chinese). The most significant challenge arises from variations in word forms. For instance, "cat" and "cats" are semantically related, but they are different tokens. **Stemming** is a linguistic method to merge such tokens, enabling better control over vocabulary size and term matching
  - **N-grams and phrases** are composite tokens where multiple words that consistently appear together form a single token. Examples include "San Francisco," "Salt Lake City," "Prime Minister," or "Thai food." While you can manually add such phrases to the vocabulary, we will explore automated methods to detect meaningful phrases or n-grams in the collection in the next chapter of this course
- In this chapter, we use words as the foundation for studying classical text retrieval methods. In the following chapters, we will delve deeper into tokenization and explore various linguistic transformations, along with newer approaches such as embeddings commonly used in generative AI applications.

In the year 1878 I took my degree of Doctor of Medicine of the University of London, and proceeded to Netley to go through the course prescribed for surgeons in the army. Having completed my studies there, I was duly attached to the Fifth Northumberland Fusiliers as Assistant Surgeon. The regiment was stationed in India at the time, and before I could join it, the second Afghan war had broken out. On landing at Bombay, I learned that my corps had advanced through the passes, and was already deep in the enemy's country. I ...

(IN) (THE) (YEAR) (1878) (I) (TOOK) (MY) (DEGREE) (OF) (DOCTOR) (OF) (MEDICINE) (OF) (THE) (UNIVERSITY) (OF) (LONDON) (',') (AND) (PROCEEDED) (TO) (NETLEY) (TO) (GO) (THROUGH) (THE) (COURSE) (PRESCRIBED) (FOR) (SURGEONS) (IN) (THE) (ARMY) ('.') (HAVING) (COMPLETED) (MY) (STUDIES) (THERE) (',') (I) (WAS) (DULY) (ATTACHED) (TO) (THE) (FIFTH) (NORTHUMBERLAND) (FUSILIERS) (AS) (ASSISTANT) (SURGEON) ('.') (THE) (REGIMENT) (WAS) (STATIONED) (IN) (INDIA) (AT) (THE) (TIME) (',') (AND) (BEFORE) (I) (COULD) (JOIN) (IT) (',') (THE) (SECOND) (AFGHAN) (WAR) (HAD) (BROKEN) (OUT) ('.') ...

- Lemmatization and linguistic transformation are essential for matching query terms with document terms, even if they have different inflections or spellings (e.g., "colour" vs. "color"). Depending on the scenario, one or several of the following methods can be used:
  - A common step is stemming. In most languages, words appear in various inflected forms based on time, case, or gender. Examples:

English:	go, goes, went, going, house, houses, master, master’s
German:	gehen, gehst, ging, gegangen, Haus, Häuser, Meister, Meisters

    As evident from the examples, the inflected forms differ significantly but essentially convey the same meaning. The concept of stemming is to reduce tokens to a common stem and utilize this stem instead. In some languages, like German, stemming is difficult due to its numerous irregular forms and the use of strong inflections ("gehen" → "ging"). In English, Porter defined a very simple algorithm to compute near-stems as explained on the next pages
  - Additionally, some languages permit compound words which can result in words of arbitrary length:

    German (law in Mecklenburg-Vorpommern, 1999-2013): *Rinderkennzeichnungs- und Rindfleischetikettierungsüberwachungsaufgabenübertragungsgesetz*
    (cattle marking and beef labeling supervision duties delegation law)

    Finnish: *atomiydinenergiareaktorigeneraattorilauhduttajaturbiiniratasvaihde*
    (atomic nuclear energy reactor generator condenser turbine cogwheel stage)

    In many cases, we aim to break down such compounds to improve the likelihood of matching against query terms. Otherwise, we might never find that German cattle law with a query like "Rind Kennzeichnung." However, breaking a compound may also alter the true meaning of tokens:

    German: Gartenhaus → Garten, Haus (ok, not too far away from the true meaning)
    German: Wolkenkratzer → Wolke, Kratzer (no, this is completely wrong)

- For English, the Porter Algorithm finds a near-stem of words. This stem is not linguistically correct but it often reduces words with the same linguistic stem to the same near-stem. The algorithm is highly efficient, and various extensions have been proposed over the years. In this context, we focus on Porter's original version from 1980:
  - Porter defines `v` as a „vocal“ if

    - it is an **A, E, I, O, U**
    - it is a **Y** and the preceding character is not a „vocal“ (e.g. R**Y**, B**Y**)

  - All other characters are consonants (`c`)
  - Let `C` be a sequence of consonants, and let `V` be a sequence of vocals
  - Each word follows the following pattern:

    $[C](VC)^m[V]$ `m` is the measure of the word

  - further:

    - `*o`: stem ends with `cvc`; second consonant must not be W, X or Y (-WIL, -HOP)
    - `*d`: stem with double consonant (-TT, -SS)
    - `*v*`: stem contains a vocal

  - The rules on the next pages establish mappings for words using the forms mentioned above. The variable `m` is utilized to prevent over-stemming of short words. Due to limited space, only a few rules are presented here. For a complete set of rules, please refer to one of the many implementations of the Porter algorithm or consult the original paper: **Porter, M.F.: *An Algorithm for Suffix Stripping*. Program, Vol. 14, No. 3, 1980**
  - There are 5 main steps with several sub-steps within each. Each (sub-)step includes a list of ordered rules to match the endings of terms. Only the first rule that matches is applied, and the algorithm proceeds to the next (sub-)step. Most sub-steps have only a few rules (less than 10) and not more than 20 rules. The JavaScript implementation comprises around 200 lines of code.

- In subsequent chapters of this course, we will explore more advanced methods

	Rule		Examples	
	Step 1			
a)	SSES	-> SS	caresses	-> caress
	IES	-> I	ponies	-> poni
	SS	-> SS	caress	-> caress
	S	->	cats	-> cat
b)	(m>0) EED	->EE	feed	-> feed
	(*v*) ED	->	plastered	-> plaster
	(*v*) ING	->	motoring	-> motor
	… +5 more rules if 2nd/3rd rule match			
c)	(*v*) Y	->I	pony	-> poni
	Step 2			
	(m>0) ATIONAL	-> ATE	relational	-> relate
	(m>0) TIONAL	-> TION	conditional	-> condition
	(m>0) ENCI	-> ENCE	valenci	-> valence
	(m>0) IZER	-> IZE	digitizer	-> digitize
	… +16 more rules			

Rule		Examples	
Step 3			
`(m>0) ICATE`	`-> IC`	triplicate	-> triplic
`(m>0) ATIVE`	`->`	formative	-> form
`(m>0) ALIZE`	`-> AL`	formalize	-> formal
... +4 more rules			
Step 4			
`(m>1) and (*S or *T)ION`	`->`	adoption	-> adopt
`(m>1) OU`	`->`	homologou	-> homolog
`(m>1) ISM`	`->`	platonism	-> platon
... +16 more rules			
Step 5			
a) `(m>1) E`	`->`	rate	-> rate
`(m=1) and (not *O)E`	`->`	cease	-> ceas
b) `(m>1 and *D and *L)`	`-> single letter`	controll	-> control

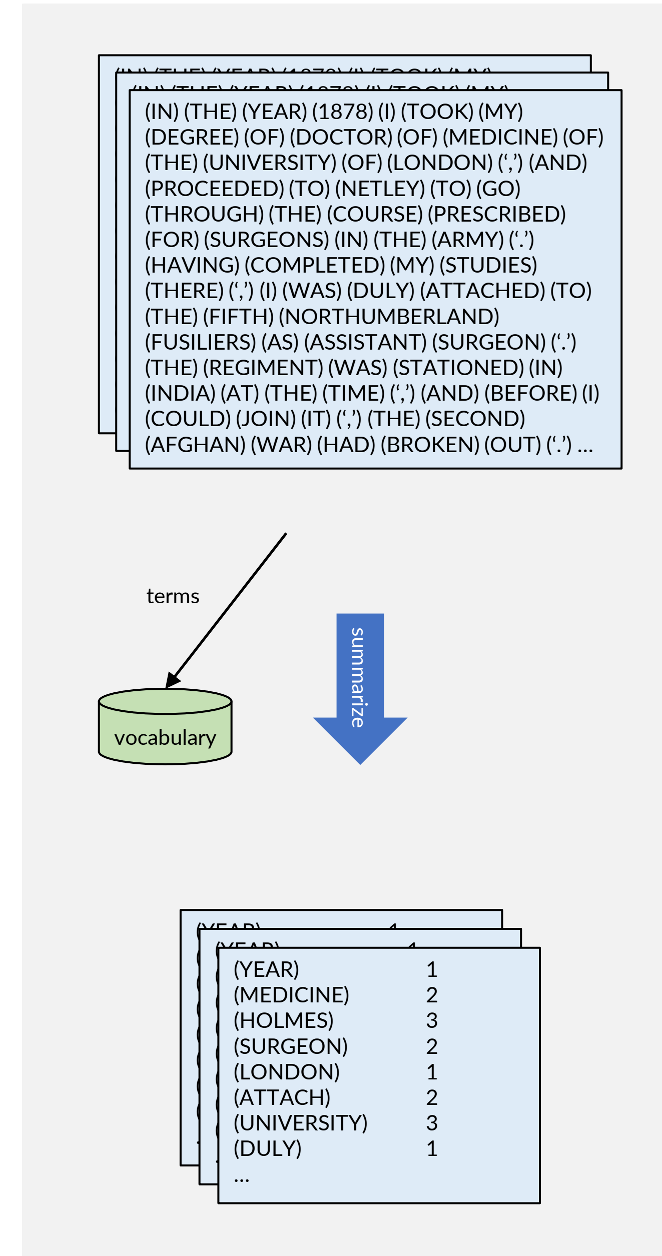
3.2.4 Action 4: Summarize

- During summarization, we create concise representations for documents, usually in the form of a high-dimensional feature vector where components represent terms and their occurrences in the documents. To achieve this, we maintain a vocabulary and assign each term a dimension in the feature space.
- To control vocabulary size, we discussed linguistic transformations in the previous action. During summarization, we also evaluate the importance of terms and their ability to describe the content of documents in the collection. The **inverse document frequency (IDF)** is a widely used method to measure the significance of terms. Additionally, we look at stop word elimination as a simpler method of discrimination.
- Once terms are extracted, classical retrieval methods generally use one of two methods to build the feature vector. Let D_i be a document, M be the size of the vocabulary. Then, $d_i \in \mathbb{R}^M$ is its feature representation, and $d_{i,j}$ represents the term t_j in the vocabulary. Additionally, we use $tf(D_i, t_j)$ to denote the number of occurrences of term t_j in document D_i.
  - The **set-of-words** model is a basic representation that only considers whether a term is present or not. It disregards the order of terms, their number of occurrences, and proximity between terms. The feature vector is binary where dimension j indicates the presence of term t_j

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

  - The **bag-of-words** model is a more common representation and differs from the set-of-words by preserving term frequencies:

$$d_{i,j} = tf(D_i, t_j)$$



- Classical retrieval models treat terms as independent regardless of how close they are syntactically or semantically. For example, "cat" and "cats" are considered different terms. This implies that a query for "cats" will not match documents containing only "cat". To make them match, we need to reduce these forms to the same term as shown with the stemming algorithm during tokenization. The same principle applies to spelling mistakes or variations, either in the document or in the query: "colour" does not match with "color".
- Controlling the vocabulary size is not primarily a storage or performance concern as we will see with the indexing methods for classical retrieval. Usually, a vocabulary can include millions of terms. However, most documents consist of only a few hundred or thousand terms, depending on how we split them. Consequently, the feature vectors are densely populated with non-zero values. Using the **inverted file** method, we store only the non-zero values and during retrieval, we only consider documents that contain a query term.
- However, we notice many terms that are grammatically necessary but do not contribute significantly to the content description. For example, the article "the" in English is one of the most frequent terms in English texts but does not provide relevant information to describe the content. Since almost all English texts contain this article, a search with "the" would retrieve all documents making it unable to differentiate between relevant and non-relevant ones.
- Apart from "the," there are other common **stop words**, as shown in the table on the right with the 50 most frequent terms in a collection D of around 20,000 documents. The second column shows the document frequency $df(t_j)$ which is the number of documents containing the term t_j shown as a percentage of all documents. The last column shows the term frequency of t_j across all documents in the collection D, presented as a percentage of the total number of terms in D (source: https://faculty.georgetown.edu/wilsong/IR/WD3.html)
  - The top-50 terms already account for one-third of all terms in the collection, yet they do not significantly contribute to the document description (wasting storage space)
  - All terms appear in more than 60% of the documents, making them unable to distinguish between relevant and non-relevant documents, as they match with most documents
- Stop word lists for most languages are readily available, for example: https://www.kaggle.com/datasets/heeraldedhia/stop-words-in-28-languages

t_j	$df(t_j)$	$tf(D,t_j)$
the	100%	6.03%
a	99%	2.57%
of	99%	2.55%
and	99%	2.40%
to	99%	2.58%
in	99%	2.00%
for	98%	0.97%
that	96%	1.15%
on	96%	0.75%
is	95%	0.93%
with	95%	0.70%
at	93%	0.55%
by	92%	0.49%
it	92%	0.69%
as	91%	0.57%
but	91%	0.49%
from	90%	0.44%
be	88%	0.46%
an	88%	0.38%
have	88%	0.45%
was	85%	0.65%
not	84%	0.38%
this	83%	0.35%
are	83%	0.45%
has	83%	0.40%
who	81%	0.37%
they	78%	0.37%
he	78%	0.70%
one	77%	0.26%
said	77%	0.70%
more	75%	0.26%
about	75%	0.27%
or	75%	0.31%
when	74%	0.24%
their	71%	0.27%
his	70%	0.49%
had	70%	0.29%
been	70%	0.21%
all	69%	0.20%
which	69%	0.20%
will	68%	0.27%
out	68%	0.20%
up	68%	0.20%
if	67%	0.21%
than	66%	0.18%
were	66%	0.22%
would	65%	0.23%
can	65%	0.20%
new	64%	0.23%
there	64%	0.18%

- Instead of manually maintaining stop word lists, a more pragmatic approach 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 c/r with a constant c that only depends on M as shown on the right side.
- The sum of all $p(r)$ equals 1 and plugging-in the $p(r) = c/r$ for all terms results in a closed formula to estimate c based on the number of terms M. For example, in a collection with $M = 5{,}000$ different terms, $c = 0.11$, while in a collection with $M = 100{,}000$, $c = 0.08$.
- The bottom right figure displays the **Zipf distribution** (blue line). As explained earlier, the most frequent words (above the upper cut-off line) hold minimal significance since they appear in nearly every text. The least frequent words (below the lower cut-off) are discriminative but unlikely to appear in queries. The range of meaningful words falls between the lower and upper cut-off points.
- Initially, the idea was to establish cut-off thresholds and exclude words beyond those limits. This would save storage space and enhance search speed. Nowadays, the common practice is to retain all terms, including stop words, but consider the terms' discriminating power (see the red line in the figure) to determine their weight during relevance assessment.
- Consider the search for "it" which is a stop word. If we were to eliminate this term, we would lose the ability to search for IT books or the book "IT" by Stephen King. A query like "the cat" would still search for both terms in documents but would assign significantly higher weight to occurrences of "cat" to determine relevance.

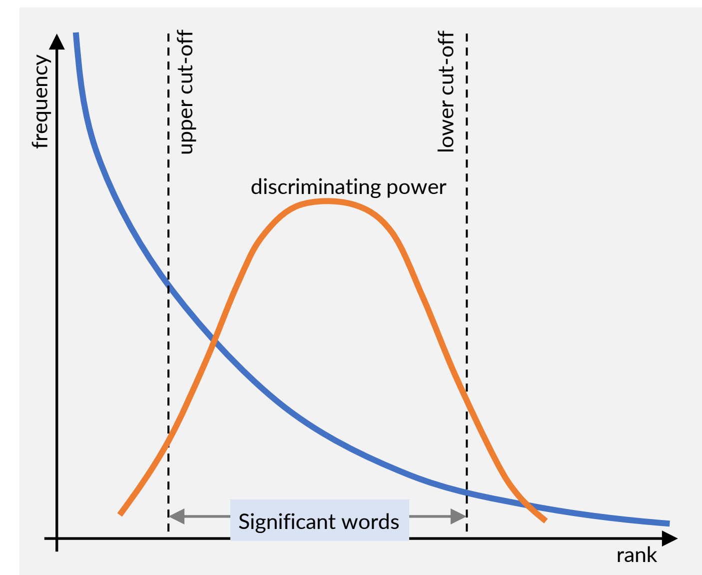
$$p_r = \frac{c}{r} = \frac{tf(t)}{N}, \quad \text{term } t \text{ with } rank(t) = r$$

c is a constant depending only on M

$$1 = \sum_{r=1}^{M} p_r = \sum_{r=1}^{M} \frac{c}{r} = c \cdot \sum_{r=1}^{M} \frac{1}{r}$$

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

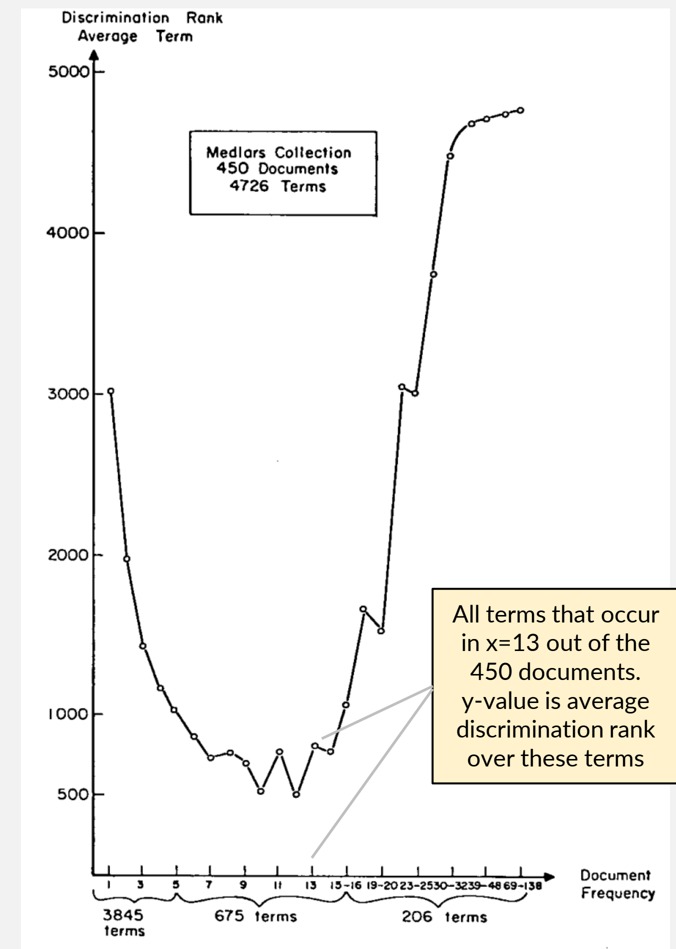
M	5'000	10'000	50'000	100'000
c	0.11	0.10	0.09	0.08



- In their 1975 paper, Salton, Wong, and Yang took a different approach by exploring methods to quantify the discriminatory power of terms. Let's consider a collection with documents D_i and the similarities between them given by $0 \leq sim(D_i, D_j) \leq 1$. We examine the collection twice, once with the term t in documents and once with it removed, to analyze the impact of the term's presence on similarities. Removing a valuable term from the collection causes documents to become more similar to each other. This is because the valuable term helped to distinguish documents, resulting in lower similarities between them.
  - Let $tf(D_i, t_j)$ represent the term frequency of term t_j in document D_i
  - We determine the centroid document C by aggregating all M terms with their average frequency $tf(C, t_j)$ across the N documents
  - Then, we define the density of the collection as the sum of all similarities between documents and their centroid C:

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

  - Finally, we compute the density Q_t for the collection without the term t, and define the discrimination power of term t as: $dp(t) = Q_t - Q$
    - $dp(t)$ is large: if we remove the term t from the collection, similarities to the centroid increase. In other words, the term t differentiates the collection and is hence a significant term
    - $dp(t)$ is negative: if term is present, documents are more similar to the centroid. This can happen, for instance, if a word occurs very frequently in all documents and thus dominates the similarity score
  - Sorting terms by their decreasing $dp(t)$-value assigns a discrimination rank to each term t. The figure on the right illustrates the average ranks (y-axis) for terms occurring in 1, 2, 3, ..., up to 138 documents.



Observation: Terms that appear in very few or numerous documents receive a high discrimination rank. However, terms occurring in 9-12 documents have the smallest discrimination ranks. These terms add significantly to the description of documents in the collection.

source: Salton, Wong, Yang (1975)

- Karen Spärck Jones (1972) introduced a statistical interpretation for term discrimination called **inverse document frequency (idf)** which has evolved into the standard method for term weighting in relevance assessment. The document frequency $df(t)$ indicates how many documents contain the term t at least once. Let N be the collection's document count. The inverse document frequency $idf(t)$ is expressed as:

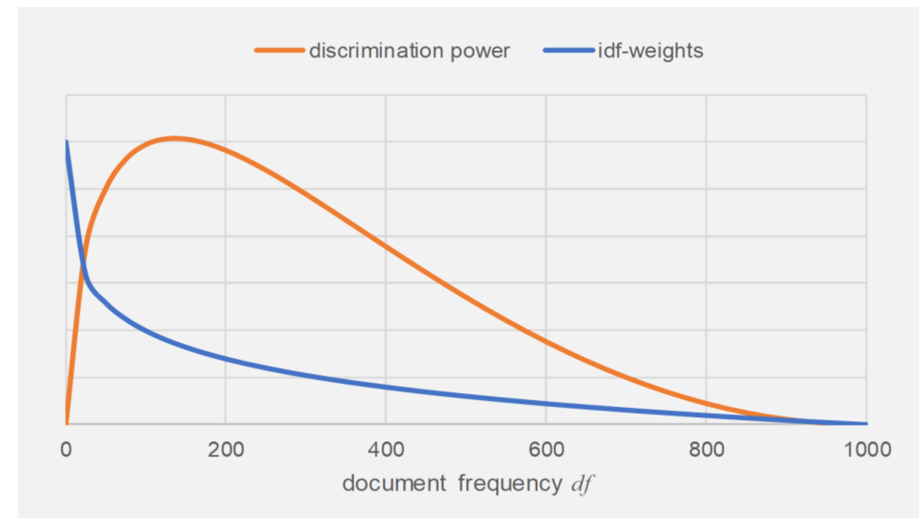
$$idf(t) = \log\frac{N+1}{df(t)+1} = \log(N+1) - \log(df(t)+1)$$

  Note that there exist many variants of the idf-formula, but all share the same structure as shown above.

- We can utilize idf to assign weights to components in both query and document feature vectors. As a simplification, let us assume that a term only occurs once in a query. Furthermore, we can estimate the probability that a term t is part of the query to be proportional to $df(t)/N$ (we need to normalize by the sum over all terms to obtain probability values). Finally, the components of the weighted document vector for D_i are given by $idf(t) \cdot tf(D_i, t)$

- Comparing vectors in vector space retrieval relies on the inner vector product. We multiply query and document components and aggregate these values. Consequently, the term's discrimination power approximately equals $idf(t)^2 \cdot tf(D_i, t) \cdot p(t)$ over all queries and documents. This value predicts a term's contribution to the relevance assessment (here for the inner vector product), or in other words, how useful the term is to describe the content and to distinguish between relevant and non-relevant documents.

- The lower right graph depicts idf-weights (blue) and discrimination power (red) based on document frequency df with $N = 1{,}000$ documents:
  - Terms with low document frequencies (left side) have high idf-weights but are scarcely present in queries, leading to low discrimination power
  - On the right side, terms with high document frequency have both low weights and discrimination power
  - Terms around $df = 100 = 0.1 \cdot N$ exhibit the highest discrimination power



3.3 Text Retrieval Models

- In the upcoming sections, we explore various retrieval models, examining their pros and cons. While we focus on the key methods, it's important to note that there are numerous extensions in literature. Throughout this chapter, we'll employ the following notations:

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 $1 \leq j \leq M$
$\boldsymbol{d}_i$	$\{0,1\}^M$, $\mathbb{N}^M$, 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 $a_{:,i} = \boldsymbol{d}_i$.
$tf(D_i, t_j)$	$\mathbb{N}$	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
$\boldsymbol{q}$	$\{0,1\}^M$, $\mathbb{N}^M$, 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

3.3.1 Standard Boolean Model

- The original Boolean models treated documents and queries as **sets of words**, aiming to find documents containing all query terms. Later, Boolean expression enhanced queries and allowed for more complex search scenarios. A key advantage was the ability to decide for each document whether it is relevant and in the result, independently of the rest of the collection. As such, the **Standard Boolean Model** functions as a filtering predicate selecting relevant items rather than assessing their relevance. Initially, Boolean retrieval focused on data retrieval, lacking the capacity to rank documents by importance. We labeled these as "Retrieval-only" engines.
- Boolean expressions consist of two atomic predicates and two methods for merging them into expressions. The atomic predicates are: 1) presence of a term ('must be present') and 2) absence of a term ('must not be present'). These atomic predicates are then combined using the AND and OR operators to create the query expression.

• $Q = t$	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 fullfilled
• $Q = Q_1 \wedge Q_2$	Both sub-query q_1 and q_2 fullfilled

- Following the rules for Boolean expressions, we can transform the query expression into a disjunction normal form:

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

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

- Query evaluation can be approached in two ways: 1) individually assess the predicate for every document, and 2) employ set operations to derive the result set from the entire collection:
  1) For each document being examined, calculate the values for all $\tau_{l,k}$ based on the presence or absence of query terms in the document, considering whether the term 'must be present' or 'must not be present'. If the evaluation of the disjunctive normal form results in a true value, the document is marked as relevant

2) To enhance query evaluation speed, we only need to focus on documents that either contain the query term ('must be present') or don't contain it ('must not be present'). Consequently, for each atomic predicate, we can create sets $\mathbb{S}_{l,k}$ that include precisely the documents that satisfy the atomic predicate:

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

Following the same structure of the disjunctive normal-form of the query, we use set intersections and unions to compute the final set of relevant documents:

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

Later in this chapter, we will introduce the inverted file method, we applies this evaluation scheme to provide fast response times.

- **Advantages:** Simple model with clear query semantics. Easy to implement and user-friendly. Fast evaluation with sets enables quick searches, even for large data sets. Boolean expressions offer precise control for including or excluding documents, influencing result size. This model can explain why a document was considered relevant. Easy to extend with other filtering criteria over metadata of documents (e.g., language = 'English')
- **Disadvantages:** Limited control over result size—users may get too few or too many results. Larger result sets lack ranking, requiring manual browsing. If the set of relevant documents is small, the method does not show 'partial matches', i.e., documents that fulfill some of the atomic predicates but not all. Although the query language is simple, users may find it hard to express a complex information need as a combination of ANDs and ORs. However, to improve the definition of 'what is relevant', users require more complex queries. All terms have the same weight, hence, stop words contribute equally to the result as the more significant terms. The Boolean model resembles data retrieval more than information retrieval. We will consider superior models with ranking that offer similar simplicity and performance.

3.3.2 Extended Boolean Model

- In 1983, Salton et. al. extended the Boolean model to overcome the drawbacks discussed previously:
  - introduce scores for ranking, considering weights for terms and term occurrences for atomic predicates
  - support partial matches, i.e., positive scores for documents that do not fulfill all atomic predicates
- The Extended Boolean Model adopts a bag-of-words approach, assigning normalized vectors ($\boldsymbol{d}_i$) to documents using term occurrences and inverse document frequency (idf). Normalization ensures values within the vector components range between 0 and 1:

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

- However, the query remains a Boolean expression as in the standard model:

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

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

- Rather than 'true' and 'false', atomic predicates yield a similarity score between 0 and 1, determined by the vector component and the 'must be present' or 'must not be present' predicate:

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

- Using the similarity scores for atomic predicates, we can establish how scores are merged for the AND and OR operators in Boolean expressions. Several common methods exist, a selection of which is provided below:
  - **Fuzzy Algebraic:** only works for two operands

$$sim(Q_1 \wedge Q_2, D_i) = sim(Q_1, D_i) \cdot sim(Q_2, D_i)$$
$$sim(Q_1 \vee Q_2, D_i) = sim(Q_1, D_i) + sim(Q_2, D_i) - sim(Q_1, D_i) \cdot sim(Q_2, D_i)$$

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

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

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

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

  - **P-Norm-Model:** distances (p-norm) in the query (sub-)vector space

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

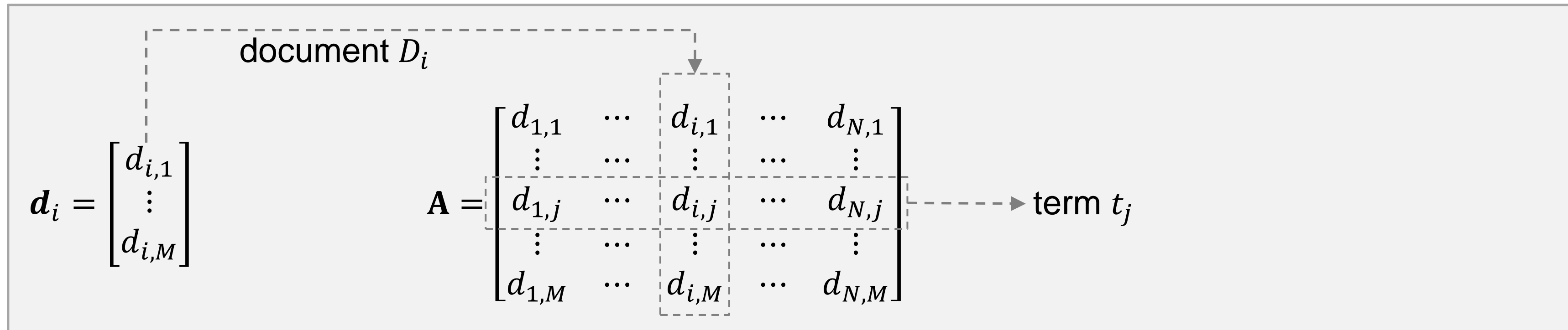
- **Advantages:** simple model with clear query semantics as with standard Boolean model. User-friendly and easy to implement. While query evaluation is heuristic, it offers solid performance. With the inverted file method, similarity values can be efficiently computed. Unlike the standard Boolean model, it provides ranked lists and partial matches, allowing control over result size. Terms are treated differently based on term occurrence and discrimination power.
- **Disadvantages:** Heuristic similarity scores lack clear theoretical explanation. Users might struggle to express complex information needs using the simple query language. Retrieval quality is decent, but other methods with similar computational complexity yield better outcomes.

3.3.3 Vector Space Retrieval

- The initial version of the vector space retrieval model was introduced in the SMART retrieval system by Salton et al. It remains the most widely used classical retrieval model, and we will explore advanced extensions and implementations in this chapter for state-of-the-art retrieval performance.
- Unlike Boolean methods, the vector space retrieval model treats documents and queries as vectors in a high-dimensional feature space. It employs vector-based similarity metrics for ranking. A document D_i is represented as a vector $\boldsymbol{d}_i$, utilizing idf-weighted term frequencies. Unlike the extended Boolean models, we refrain from normalizing the term frequencies.

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

- All document representations can be merged into the term-document matrix $\mathbf{A}$. Each column in $\mathbf{A}$ corresponds to a document, and each row represents a term in the vocabulary. Hence, matrix element $a_{j,i} = d_{i,j}$, following the convention of addressing matrix elements by rows and then columns.



- While we illustrate the method in this chapter using the term-document matrix and outline matrix-vector operations for score computation, practical implementations do not store or utilize matrix calculations due to the matrix's sparsity, where many elements are 0 as documents usually have only a few terms. We will explore more efficient evaluation techniques in the subsequent parts of this chapter.

- Queries are depicted as sparse vectors, denoted as $\boldsymbol{q}$. Unlike Boolean expressions, a query is treated as a mini-document or search prompt, following identical processing steps and vocabulary use as documents. This results in term frequencies in queries, yielding a component q_j through the following method:

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

- Various methods exist to compare document vectors with query vectors. In this context, we will discuss the most prominent ones:
  - The **inner vector product** uses the dot-product between the query and document vector. When applied to the entire collection, we multiply the term-document matrix by the query vector and then rank documents based on decreasing similarity values. It is important to note that similarity here is not confined to a range between 0 and 1, and literature often refers to it as retrieval status value (RSV):

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

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

    The formula shows that only query terms impact the similarity score, with terms absent in the query yielding a value of 0 for $q_j \cdot d_{i,j}$, irrespective of their frequency in documents. In contrast, documents with larger $d_{i,j}$ values for query terms, that is more term occurrences, receive higher ranks. Notably, significant terms with higher idf values have more influence, and this influence is amplified due to idf weighting in both queries and documents. Finally, we observer the 'partial-match' capability of the model. If a document shares at least one term with the query, then the score is positive.
  - The **cosine measure** calculates the angle between document and query vectors. It implies that documents need to contain query terms for high scores. Absence of query terms widens the angle between the vectors, leading to lower scores.

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

Similar to the inner vector product, scores for all documents can be calculated through matrix-vector multiplications. For this, we normalize the query vector by its size and introduce a diagonal matrix $\mathbf{L}$ with inverse document lengths to dynamically normalize document vectors.

$$\boldsymbol{sim}(Q,\mathbb{D}) = \begin{bmatrix} sim(Q,D_1) \\ \vdots \\ sim(Q,D_N) \end{bmatrix} = \mathbf{L}\mathbf{A}^\top \boldsymbol{q}' \quad \text{with} \quad \mathbf{L} \in \mathbb{R}^{N\times N} = \begin{bmatrix} \frac{1}{\|\boldsymbol{d}_1\|} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{\|\boldsymbol{d}_N\|} \end{bmatrix} \quad \text{and} \quad \boldsymbol{q}' = \frac{\boldsymbol{q}}{\|\boldsymbol{q}\|}$$

Alternatively, we could normalize document and query vectors during the extraction step and save normalized versions. This makes the inner vector product and the cosine measure equivalent since vectors have a length of 1. Additionally, similar to the inner vector product, partial matching capability is achieved, and terms absent from the query do not affect the search order. However, the cosine measure is less affected by term occurrences compared to the inner vector product due to normalization.

- For a simplified visualization of vector space retrieval, documents are projected into the smaller query vector space spanned by the query terms, while other dimensions have no effect on search order:
  - Using the inner vector product, a hyperplane through the origin is established with the query vector as its normal. Documents farther from this plane are considered more relevant
  - On the other hand, the cosine measure creates hyper-cones with the query vector as their axis. Higher cosine values correspond to smaller angles of a hyper-cone embedding the document
  - Documents lacking query terms are placed at the origin, yielding a value of 0 with both measures. This allows us to disregard such documents and focus on those containing at least one query term. This leads to efficient retrieval methods explored later using inverted files.
  - An issue arises when query terms are similar (e.g., 'house' and 'villa'), as they might not affect results unless pre-processing merges them. This limitation is common in classical retrieval techniques, often addressed by automatically expanding queries with related terms.

- **Example:** Let's examine a simple collection of three documents to understand the method:

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 extract terms, find document frequencies and inverse document frequencies. The document and query are represented as vectors ($N = 3$, $M = 11$) as follows:

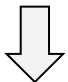
j	Term t_j	$df(t_j)$	$idf(t_j)$	d_1	d_2	d_3	q
1	a	3	0				
2	arrived	2	.176		.176	.176	
3	damaged	1	.477	.477			
4	delivery	1	.477		.477		
5	fire	1	.477	.477			
6	**gold**	2	.176	.176		.176	.176
7	in	3	0				
8	of	3	0				
9	**silver**	1	.477		.954		.477
10	shipment	2	.176	.176		.176	
11	**truck**	2	.176		.176	.176	.176

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

$\mathbf{A}$ (columns d_1, d_2, d_3)

with inner vector product

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



$D_2 > D_3 > D_1$

- **Advantages:** Extremely simple and intuitive query model. Term weights have a good impact on the scores and differentiate between query terms, e.g., reducing the impact of stop words in the query. Easy to implement and highly efficient in calculation. Outperforms Boolean models and can rival top retrieval methods. Naturally supports partial match queries, and documents do not have to include all query terms for 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). If documents are of different lengths, scores can vary significantly due to the higher term occurrences in larger documents. Main assumption of retrieval model is independence of terms which may not hold true in typical scenarios (see synonyms and homonyms).

3.3.4 Probabilistic Retrieval

- The primary criticism of the existing models lies in their heuristic nature. While they perform well, their correctness lacks a solid foundation. Probabilistic retrieval provides a formal approach based on probabilities. $P(R|D_i)$ is the probability that a document D_i is relevant for a query Q, and $P(NR|D_i) = 1 - P(R|D_i)$ is the probability that it's not relevant. The similarity value between query Q and document D_i is then 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 straightforward approach grounded in several key assumptions for calculating the mentioned conditional probabilities. These assumptions are as follows:

  1. Term frequency does not matter (utilizing a set-of-words document model)
  2. Term independence (consistent with previous models)
  3. Terms absent from the query do not influence ranking (if a term is absent from the query, it's assumed to be equally distributed among relevant and non-relevant documents)

- Given these assumptions, our next step is to derive a closed formula for the similarity scores. To begin, we apply Bayes' theorem to the conditional probabilities above:

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

  These new probabilities can be interpreted as follows: $P(R)$ and $P(NR)$ represent 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 document D_i belongs to the set of relevant and non-relevant documents, respectively.

- Now, leveraging the assumption of binary document vectors and term independence:

Assumption 1: Documents are binary vectors

Assumption 2: Terms are independent

Assumption 1: Documents are binary vectors

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

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

- Let's use a compact notation for the conditional probabilities in the formula above. Define:

$$r_j = P(d_{i,j}=1|R) \qquad\qquad n_j = P(d_{i,j}=1|NR)$$

r_j is the probability of a relevant document having the term t_j, and n_j is the probability of a non-relevant document having the term t_j. Using this notation, we can express the similarity value in simpler terms:

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

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

It is important to observe that there is no need to calculate $P(R)$ and $P(NR)$ as they are solely determined by the query and do not affect the document ranking as they linearly scale the similarity values. Therefore, the simplified lower formula produces the same document ranking as the original upper formula.

- We conclude by applying the third assumption: if term t_j is absent in the query, we assume $r_j = n_j$, i.e., non-query terms occur with equal probability in relevant and non-relevant documents. As a result, when $q_j = 0$, the ratios r_j/n_j and $(1-r_j)/(1-n_j)$ become 1, and can be omitted from the calculations:

Assumption 3: non-query terms do not impact result

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

We remove the condition $d_{i,j} = 1$ from the second product and need to adjust in the first product:

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

Next, we remove the second product, which solely depends on the query, and linearly scales the similarity values:

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

Finally, we arrive at a simple similarity function as a sum of c_j-values. It is important to note that we only need to calculate c_j for query terms, which as with other models so far greatly boost query evaluation with inverted files:

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

- To calculate the c_j-values, the BIR model starts with initial estimates for a first result list, and then refines these estimates based on user feedback to enhance the results. With ongoing user input on relevant and non-relevant documents in the outcomes, we can iteratively adjust the estimates and offer better outcomes.
  - We introduced r_j and n_j as the probabilities that a relevant and non-relevant document contains the term t_j, respectively. With the user's relevance assessment, we now possess subsets of relevant and non-relevant documents, which allow us to estimate these probabilities by counting the occurrences of term t_j in these subsets
  - **Initial Estimates:** In the absence of feedback, we assume that query terms are more likely to appear in relevant documents, and in non-relevant documents they follow their document frequency. The following estimates are used initially to compute the c_j-values (n_j includes smoothing)

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

  - **Estimates with Feedback:** in each iteration, we ask the user to rate the K retrieved documents and annotate them with relevant (R) and non-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 by counting:

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

    We employ the values 0.5 and 1 in the formula above to avoid numerical problems (0-divisions). When no feedback is given, with $L = l_j = 0$, we can set $K = N$ and $k_j = df(t_j)$ to justify the initial estimates.
  - The more user feedback we gather, the more accurate the estimates for r_j and n_j become. However, users might be reluctant to provide feedback.

- **Advantages:** The BIR model establishes similarity values on a probabilistic basis through basic assumptions. Document ranking depends on the likelihood of being relevant for the query. Only query terms are necessary for similarity calculations, and the inverted file method offers efficient evaluation. The model performs well, especially after some feedback iterations. It also accommodates partial match queries, where not all query terms need to appear in relevant documents.
- **Disadvantages:** The basic assumptions of the BIR model may not always be valid. As mentioned in the vector space model, term independence is not universally applicable. More complex probabilistic models address term dependence, but they can bring extra computational complexity. Additionally, the document ranking in BIR doesn't consider term frequencies or the discrimination power of terms. Finally, not all users are willing to assist the system with feedback to improve the search results.

3.3.5 Okapi Best Match 25 (BM25)

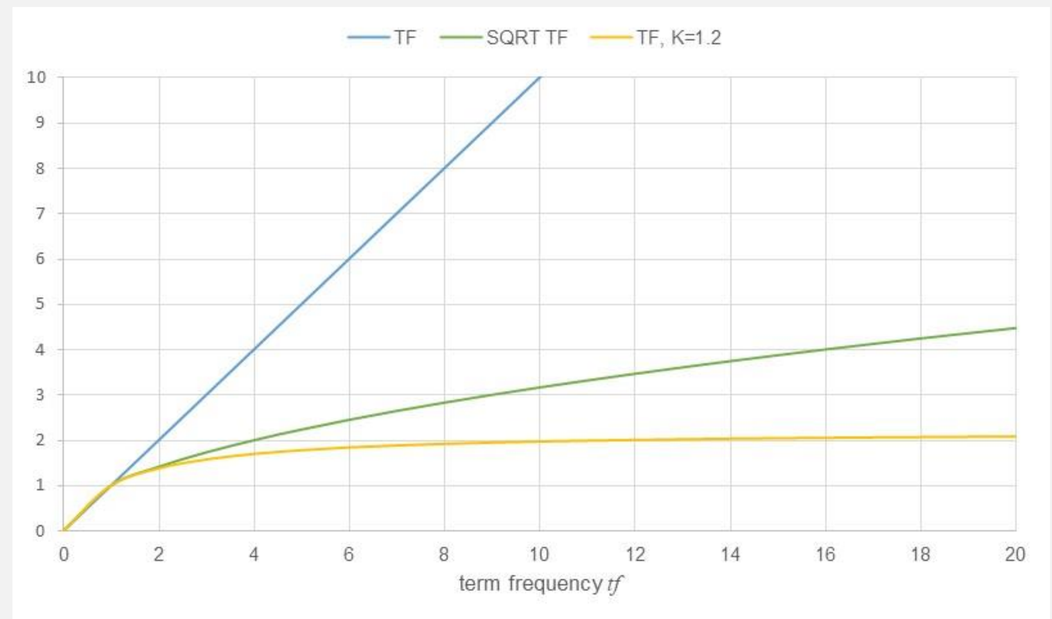
- The Okapi BM25 ranking function was developed at London's City University and is rooted in Karen Spärck Jones' probabilistic framework from the 1970s and 1980s. It is notably applied in Lucene, the engine behind Solr, Elasticsearch, and OpenSearch—three widely used systems for observability, security analytics, and full-text search. BM25 builds on the vector space model as discussed before enhancing it with a probabilistic approach to enhance relevance evaluation.
- Some limitations in the previously discussed models stem from heuristic approaches to identify relevant documents. Researchers developed better frameworks for relevance assessment, driven by key observations:
  1. **Query Term Significance:** the presence or absence of query terms is crucial for relevance assessment
  2. **Partial Matches:** not all relevant documents contain every query term
  3. **Document Length:** longer documents have more terms, but shorter relevant ones should score well too
  4. **Term Specificity:** rare words often carry more meaning than common ones
  5. **Term Saturation:** while term frequency matters, overly frequent terms should not dominate
  6. **Fine Tuning:** flexibility to adjust ranking based on search context
  7. **Efficiency:** efficient retrieval and relevance assessment are essential
  8. **User Feedback:** if available, integration of user feedback for improved search quality
  9. **Term Proximity:** closeness of query terms in a document may indicate higher semantic relevance
  10. **Term Dependence:** recognizing term dependencies, like matching query 'animals' to 'cats' or 'dogs' in documents
- BM25 addresses these observations or provides ways to consider them. We will cover **Efficiency** in the upcoming section on indexing structures and explore **Term Proximity** and **Term Dependence** in the next chapter, where we delve into natural language processing methods.

- Term frequencies play a crucial role in determining document relevance. Typically, we assume that a document's relevance is linked to the frequency of query term occurrences within it. This notion lead to the creation of the $tf \cdot idf$ vector component description. Nonetheless:
  - A document with the search term 'cat' occurring a hundred times is certainly relevant, but it should not be considered twice as relevant as a document with 50 occurrences of 'cat'. In essence, the linear factor tf exaggerates the relevance. It also makes the method vulnerable to spamming attacks
  - Shorter documents have fewer occurrences of terms compared to much longer documents. However, they can be equally or even more relevant. Yet, the $tf \cdot idf$ scheme tends to favor longer documents with higher term frequencies. Very long documents covering a broad range of topics may appear relevant due to their numerous occurrences but users may find it difficult to easily extract the relevant pieces
- A simple adjustment like using $\sqrt{tf}$ instead of tf does not provide significant improvement. We require a function that levels off after a certain occurrence threshold. With $\sqrt{tf}$ we could still influence scoring with excessive spamming of potential query terms.
  - Initially disregarding document length, we can saturate term frequencies as follows:

$$\widehat{tf}_k = \frac{tf \cdot (k+1)}{tf + k}$$

(k+1) scales values but does not impact ranking

  - Typically, $k \in [1, 2]$ with Lucene using $k = 1.2$
  - As depicted in the figure on the right, the updated values $\widehat{tf}_k$ saturate relatively swiftly to the value 2.2 with k=1.2, whereas unsaturated tf and $\sqrt{tf}$ values increase without limit
  - In essence, k serves as a hyperparameter that enables adjustment of the impact of term occurrences on the scoring
  - Note: Lucene uses $tf/(tf + k)$ omitting the scaling factor $(k + 1)$ in the numerator



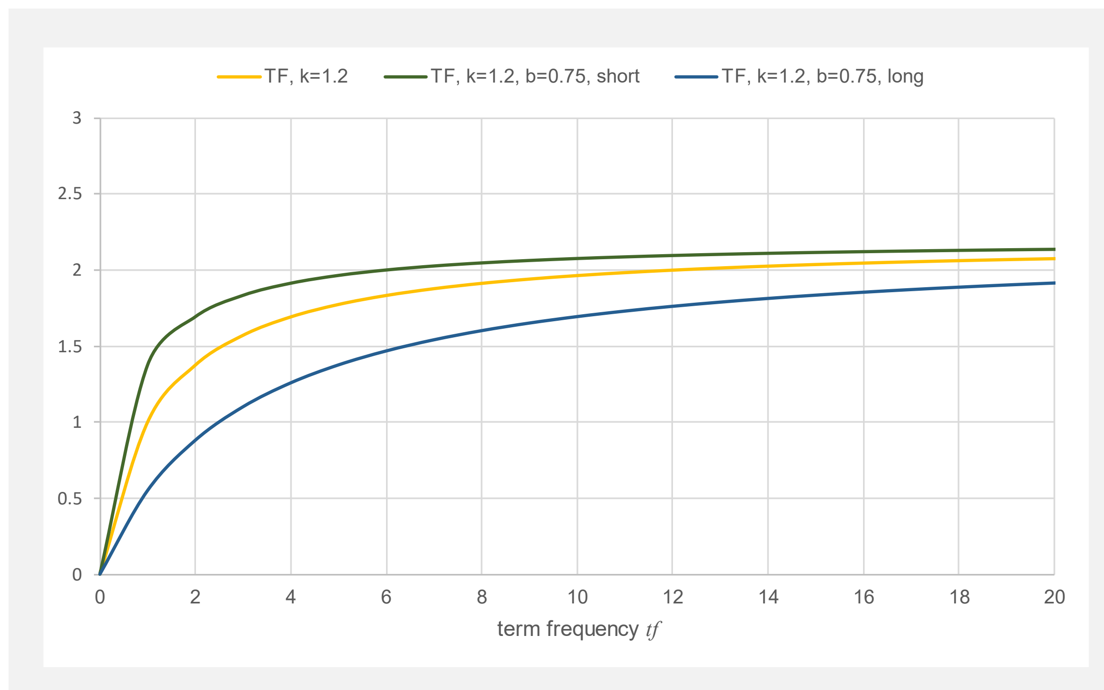
- Now, let's examine document length. Lengthier documents include more terms and should saturate at a slower rate than shorter ones that might not have as many terms. The cosine measure tackled this by normalizing vectors by their length, and then utilizing the inner vector product to determine the angle between the vectors, which remains unaffected by document length. However, BM25 takes a different approach. It employs a summation across all query terms, similar to the inner vector product, while modifying the core formula to account for document length:

$$\widehat{tf}_k(D) = \frac{tf \cdot (k+1)}{tf + k \cdot \left(1 - b + b\frac{|D|}{adl}\right)}$$

adl is the average document length

with $b = 0.75$ (adjustable), $|D|$ the length of document D, and adl the average length of documents in the collection

  - If $|D|$ is smaller than adl (short document), then $\left(1 - b + b\frac{|D|}{adl}\right) < 1$ and values $\widehat{tf}_k(D)$ saturate faster
  - If $|D|$ is large (long document), then $\left(1 - b + b\frac{|D|}{adl}\right) > 1$ and values $\widehat{tf}_k(D)$ saturate slower
  - $b \in [0,1]$ is a new hyperparameter that steers the impact of document length. Higher values prefer shorter documents
  - In the plot to the right, we compare $\widehat{tf}_k$ (graph in the middle) with $\widehat{tf}_k(D)$ of a short document (graph at the top) and with $\widehat{tf}_k(D)$ of a long document (graph at the bottom)
  - The difference between shorter and longer documents is significant at lower frequencies but soon diminishes as values saturate to 2.2 for $k = 1.2$
  - adl does not have to be the accurate average length of documents. Rather, we can consider it as another hyperparameter to define what 'long' / 'short' means



- We previously discussed idf-weights without providing a rationale for using that specific formula. BM25 approaches term weighting probabilistically. Previously, we derived a term weighting function using the BIR-model:

$$c_j = \log\frac{r_j \cdot (1-n_j)}{n_j \cdot (1-r_j)} \qquad r_j = \frac{l_j+0.5}{L+1} \qquad n_j = \frac{k_j-l_j+0.5}{K-L+1} \qquad \forall j: q_j = 1$$

  - We introduced r_j and n_j as the probabilities of the term t_j occurring in relevant and non-relevant documents based on user-provided relevance feedback. The calculation of r_j takes into account the number of relevant documents (l_j) out of the L retrieved ones that contain the query term, while n_j considers the number of non-relevant documents ($k_j - l_j$) out of the $K - L$ retrieved ones that contain the query term.
  - The BIR model summed up c_j values for the binary document representation. However, c_j values can also serve as weights for terms in the vector space model. We achieve this by incorporating r_j and n_j into the c_j formula:

$$c_j = \log\frac{r_j \cdot (1-n_j)}{n_j \cdot (1-r_j)} = \log\frac{\frac{l_j+0.5}{L+1} \cdot \left(1-\frac{k_j-l_j+0.5}{K-L+1}\right)}{\left(1-\frac{l_j+0.5}{L+1}\right) \cdot \frac{k_j-l_j+0.5}{K-L+1}} = \log\frac{\frac{l_j+0.5}{L+1} \cdot \frac{K-L+1-k_j+l_j-0.5}{K-L+1}}{\frac{L+1-l_j-0.5}{L+1} \cdot \frac{k_j-l_j+0.5}{K-L+1}} = \log\left(\frac{l_j+0.5}{L-l_j+0.5} \cdot \frac{K-L-k_j+l_j+0.5}{k_j-l_j+0.5}\right)$$

  - When user feedback is absent, L and l_j are 0, and we assume that all documents are non-relevant (until proven otherwise) and assign $K = N$ (number of documents) and $k_j = df(t_j)$ (documents containing the term). Substituting these values into the c_j formula results in:

$$c_j = \log\left(\frac{0+0.5}{0-0+0.5} \cdot \frac{N-0-df(t_j)+0+0.5}{df(t_j)-0+0.5}\right) = \log\frac{N-df(t_j)+0.5}{df(t_j)+0.5}$$

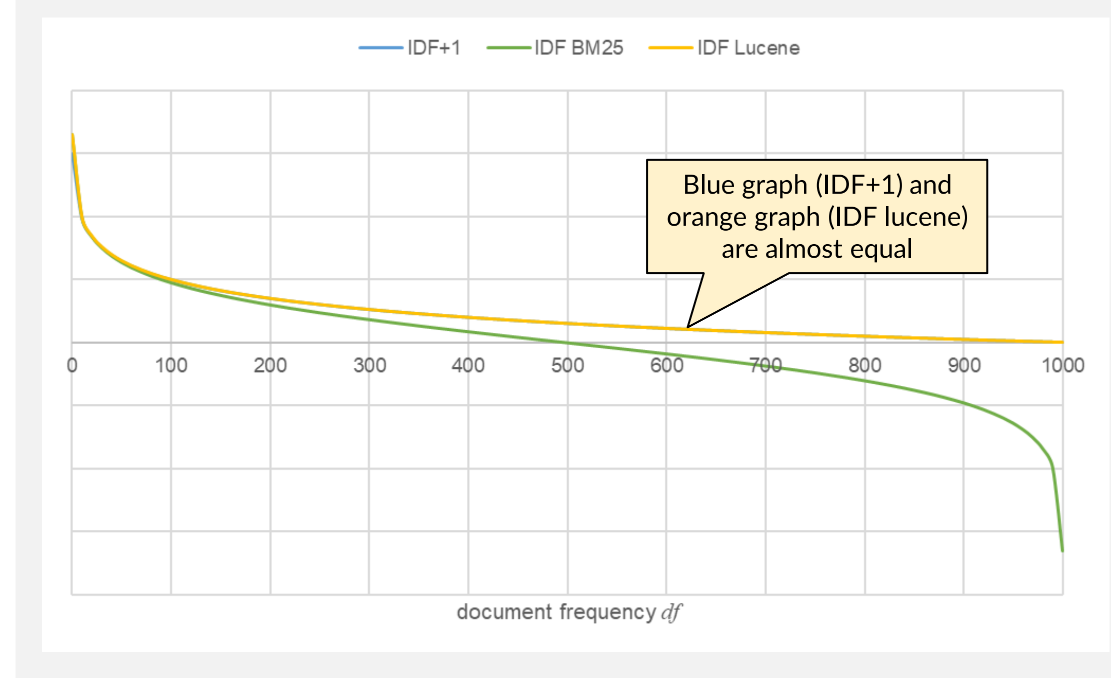
  - These c_j values are then used by the BM25 model to refine the initial idf-values we discussed earlier. Note that for terms t_j that appear in over 50% of the documents, the logarithm yields a negative value.

$$idf_{BM25}(t_j) = \log\frac{N - df(t_j) + 0.5}{df(t_j) + 0.5}$$

- Let us compare the original idf-function (IDF+1) with this new one. The graph on the right displays them against the document frequency in a collection of 1000 documents.
- Additionally, we included the idf-function used by Lucene, which incorporates a '+1' term in the logarithm to avoid negative idf-values:

$$idf_{lucene}(t_j) = \log\left(1 + \frac{N - df(t_j) + 0.5}{df(t_j) + 0.5}\right)$$
$$= \log\left(\frac{N+1}{df(t_j) + 0.5}\right)$$

  This new idf-function, however, yields almost the same values as the original IDF+1 method
- Alternatively, we can avoid negative idf-values by assigning a small positive idf value to very frequent terms (mostly stop words)



- Finally, BM25 calculates a score for a query Q and a document D_i by summing up the adjusted tf-idf values across all query terms q_j:

$$sim_{BM25}(Q, D_i) = \sum_{j=1}^{M} \log\left(\frac{N - df(t_j) + 0.5}{df(t_j) + 0.5}\right) \cdot \frac{tf(D_i, t_j) \cdot (k+1)}{tf(D_i, t_j) + k \cdot \left(1 - b + b\frac{|D_i|}{adl}\right)}$$

  - Unlike the vector space retrieval model, the idf-values are applied only once and query term frequency is not considered. Later in this chapter, we will examine Lucene's scoring function, which expands the above formula with extra components, including query term frequencies and additional term and document weighting.
  - In this fundamental formulation, BM25 encompasses three hyperparameters (k, b, adl) that allow fine-tuning the scoring function to match the requirements of the search context.

Observation	How BM25 addresses it	Remarks
Query Term Significance	The vector space retrieval model scores relevance based solely on query terms. More query terms in a document result in higher scores	Because of assumptions about term independence, query terms might not align with semantically relevant terms in the document
Partial Matches	The vector space retrieval model supports partial matches. It ranks partial matches based on the importance of the matched terms, determined by term weights	As above
Document Length	Term saturation varies based on document length; longer documents need a higher number of term occurrences compared to shorter ones	Long documents still face the challenge of ignoring query term positions. Whether query terms appear together in a paragraph is not considered. An effective solution is to divide documents into smaller sections, addressing this concern
Term Specificity	Resolved through the enhanced IDF-based weighting of the relevance scoring function	Specificity varies with context. For instance, consider the query 'car jaguar' where both terms are relatively common. However, in the context of cars, 'jaguar' is much less common than in a broader context
Term Saturation	Implemented using a saturation function on term frequencies which tackles problems related to keyword spamming and prevents excessively frequent terms, like stop words, from dominating the scoring	It is crucial to balance term specificity and term saturation to achieve the best possible outcomes in a search context
Fine Tuning	Offers various hyperparameters for tuning the ranking according to specific search scenarios; default settings are effective in many cases	Refer to discussions on training machine learning methods while validating hyperparameters
Efficiency	The scoring function relies solely on query terms. Given that queries often have less than 5 terms, inverted files ensure high performance	When employing embeddings and vector search, we exchange efficiency for improved semantic relevance evaluation
User Feedback	IDF weights of terms can be fine-tuned using relevance feedback, as demonstrated in the BIR model. Even if the implementation lacks direct support for relevance feedback, we can still modify term weights to adjust scoring based on the feedback	A simple yet effective approach to integrate feedback is through automatic query expansion. Using relevance feedback, additional terms are included that appear frequently in relevant documents but are less common in non-relevant ones
Term Proximity	BM25's relevance scoring lacks direct support for term proximity since it lacks access to term locations within documents	An important scenario involves bi-grams and tri-grams like 'New York' or 'Salt Lake City.' We can enhance our pre-processing to detect common n-grams, which we will study into in the next chapter
Term Dependence	BM25's relevance scoring lacks direct support for term dependence since it treats terms as independent of each other	Common problems involve spelling errors or synonymous forms that convey the same meaning. We will study more advanced approaches in the next chapter

3.4 Indexing Structures

- In all traditional retrieval models discussed so far, we noticed that the scoring functions rely only on the query terms. While this does not capture semantic similarity like 'cat' vs. 'animal', it's a practical trade-off for faster query processing, as we explain in this section.
- Let us assume that we have N documents, a vocabulary of M terms, documents with an average K distinct terms, and queries with an average of 5 distinct terms. Documents are modeled as sparse, M-dimensional vectors, using bag-of-words or set-of-words methods. A basic storage approach would need $N \cdot M$ entries. In the set-of-words model, an entry uses 1 bit, while a bag-of-words model takes 4, 8, 16, or 32 bits for term frequencies or $tf \cdot idf$ values. For instance, BM25 employs term saturation. Instead of storing full-precision term frequencies (16/32 bits), compression via 4 or 8-bit quantization is possible. This works because high frequencies around 100 yield similar $\widehat{tf}_k$-values, minimizing the impact of quantization errors on the search order.
- Retrieval using this simple storage approach scales linearly with collection and vocabulary size as we scan through all the data. Since that vectors are sparse with only K/M non-zero components, we mostly read 0-values that have no impact on relevance assessments. An improvement is to store a sparse representation, keeping an average of K terms per document. This totals $N \cdot K$ entries, each holding a term ID for set-of-words, and term ID with term frequencies/$tf \cdot idf$ for bag-of-words. Term identifier size varies, consuming 16 to 64 bits based on vocabulary size choice and precision for term frequencies/$tf \cdot idf$.
- Although storage consumption is much lower, we still have to search through all data to identify the best matches. During this process, most data that we read is not considered by the scoring functions as out of the K average terms stored per documents only the query terms can influence relevance assessment.
- Let's revisit the term-document matrix in vector space retrieval. The concept of the **inverted files** method, also called inverted index, is to store rows with data about which documents hold the term linked to those rows, rather than storing columns with the terms used by a document. Using sparse row encoding retains $N \cdot K$ entries, but replacing term IDs with document IDs. However, the major enhancement is during search: since only query terms impact scoring, we only read rows corresponding to query terms to produce the answer. If we have $N \cdot K/M$ documents per term on average and L query terms, we read $N \cdot K \cdot L/M$ entries, improving search by L/M. For instance, with $L = 5$ query terms and a vocabulary size of $M = 1{,}000{,}000$, we cut search time by $5/1{,}000{,}000$ (assuming average query term distribution)

3.4.1 Inverted Files for Boolean Retrieval Models

- Keeping this fundamental concept in mind, let's start with the Boolean retrieval model. The inverted index consists of the **vocabulary** (M terms), and for each term, a list of postings contains all documents that include the term. For the set-of-words model, term frequencies are not necessary, and the Boolean model does not require document frequencies or idf-values. The inverted index further contains a document table with additional metadata:

ID	Name	URL	Date	... more data
1	Paris	http://xyz.com/Paris.html	2005-01-04	...
2	Geneva	http://xyz.com/Geneva.html	2005-03-08	...
3	Milano	http://xyz.com/Milano.html	2005-04-23	...
4	New York	http://xyz.com/NewYork.html	2005-05-30	...
...				...
N	Tokyo	http://xyz.com/Tokio.html	2023-05-19	...

ID	Term	Postings [Doc-ID]
1	dog	[1, 3, 4, 6, 9, 10, 13, 21, 22, 23, 29, 30, 39]
2	cat	[4, 5, 12, 13, 14, 15, 20, 22, 30, 34, 37]
3	horse	[6, 10, 11, 14]
4	rabbit	[12, 15, 35]
…		
N	bird	[2, 3, 8, 15, 26, 35, 36]

  - As we add new documents to the table, we continue including the document ID in the postings of terms found in the document. If documents are added sequentially, the postings are arranged based on the order of document insertion, which, in our simple example, corresponds to increasing document IDs. For certain implementations, preserving this order is crucial for faster retrieval.

- The basic implementation stores postings as sets of document IDs within a vocabulary using terms as keys. For instance, `index['cat']` contains the set of IDs of documents that contain the term ‘cat’ at least once.
  - For query evaluation, we adhere to three rules:
    - expr1 AND expr2: translates to an intersection of the sets from sub-expressions expr1 and expr2
    - expr1 OR expr2: translates to a union of the sets from sub-expressions expr1 and expr2
    - expr1 AND NOT(expr2): translates to a subtraction of the set of expr2 from the set of expr1

    Generalization to AND/OR over multiple sub-expressions are straightforward
  - However, we cannot evaluate OR-queries when one sub-expression is of the form NOT(expr). While it's technically possible to construct NOT(expr) by using all documents except those returned by expr, this approach becomes inefficient for large collections
  - In AND-queries, NOT(expr)-parts need to be re-ordered to the end to apply set subtraction. Additionally, at least one element of the AND-query must not be in the form NOT(expr)
  - Indeed, while these limitations may be viewed as constraints in our implementation, they have minimal impact on practical scenarios. Queries like "cat OR NOT(dog)" do not align with typical search intentions as they essentially select all documents except those with dog but not cat, i.e., it can be rephrased as "NOT(dog AND NOT cat)".

```
cat = index['cat’]
↪[4, 5, 12, 13, 14, 15, 20, 22, 30, 34]

dog = index['dog’]
↪[1, 3, 4, 6, 9, 10, 13, 21, 22, 23, 29, 30]

horse = index['horse’]
↪[6, 10, 11, 14]

bird = index['bird’]
↪[2, 3, 8, 15, 26, 35, 36]

# cat AND dog
cat & dog
↪[4, 13, 22, 30]

# horse OR bird
horse | bird
↪[2, 3, 6, 8, 10, 11, 14, 15, 26, 35, 36]

# cat AND NOT(dog)
cat - dog
↪[5, 12, 14, 15, 20, 34]

# (cat AND dog) OR (horse AND cat AND NOT(bird))
(cat & dog) | (horse & cat - bird)
↪[4, 13, 14, 22, 30]

# (cat OR dog) AND (horse OR bird)
(cat | dog) & (horse | bird)
↪[3, 6, 10, 14, 15]

# (cat OR dog) AND NOT(horse OR bird)
(cat | dog) - (horse | bird)
↪[1, 4, 5, 9, 12, 13, 20, 21, 22, 23, 29, 30, 34]
```

- The set-based evaluation from before does not scale well with the number of documents. In cases with millions of billions of postings for a term, we want to fetch data from an external storage device (which is also a good idea for persistence). But instead of reading all postings into main memory, we read them as streams sorted by the document IDs. Take the postings of cat and dog as an example:

term	postings
`cat`	`[1, 4, 8, 10]`
`dog`	`[3, 4, 10, 12]`

  - To evaluate a query like **"cat AND dog"** we retrieve the initial entry for each term—`1` for cat and `3` for dog. If they match, the corresponding document fulfills the condition. If not, we proceed by reading the next entry for the term with the smallest document ID. In our example, we proceed to the next cat posting, which is 4. Since it does not match, we then advance to the postings of the term 'dog,' which currently has the smallest value. The subsequent dog posting is also 4, matching the cat posting. Thus, we locate our first document and return it.

    For the next result, we continue fetching subsequent postings for both terms and repeat the process. Eventually, we identify `10` as the second answer. Then, we fetch the next posting for both terms. However, as cat's postings are exhausted, we conclude the evaluation and halt iteration (even though dog still has postings, the exhaustion of cat postings indicates that any remaining document cannot match). The diagram below illustrates this approach:

step	cat (next)	dog (next)	action
1	`1`	`3`	no match, progress cat
2	`4`	`3`	no match, progress dog
3	`4`	`4`	match, return 4 as result, and progress both cat and dog
4	`8`	`10`	no match, progress cat
5	`10`	`10`	match, return `10` as result, and progress both cat and dog
6	-	`12`	stop iteration as all cat postings are visited; remaining postings in dog cannot fulfill predicate

  - The **OR-operator** is implemented similarly; however, the iteration returns each time the smallest entry of sub-expressions. In the provided example, the OR-operator would start by returning `1`, then advance cat and return `3`, progress dog and return 4, move both cat and dog and return `8`, advance cat and return `10`, move again both cat and dog, and finally return 12. The evaluation concludes once all postings are consumed.

- The evaluation of **"cat AND NOT(dog)"** evaluation follows the same pattern as the AND flow, but the outcomes differ (matching occurs when cat posting is not equal to dog posting):

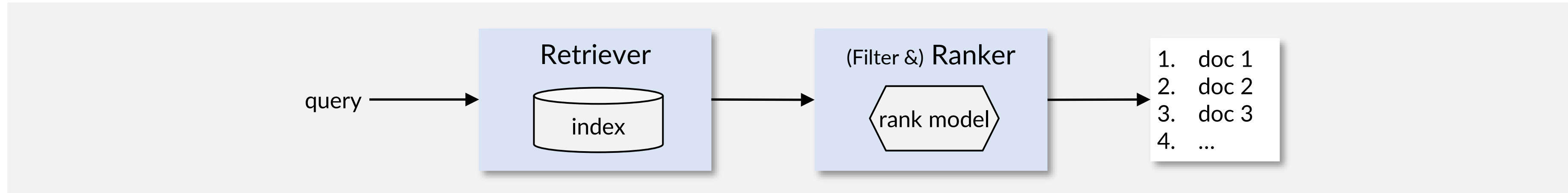
step	cat (next)	dog (next)	action
1	1	3	match, return 1 as result, and progress cat
2	4	3	match but cat is not smallest, so we progress dog
3	4	4	no match as both have the same value, so we progress both cat and dog
4	8	10	no match, return 8 as result, and progress cat
5	10	10	no match as both have the same value, so we progress both cat and dog
6	-	12	stop iteration as all cat postings are visited; remaining postings in dog cannot fulfill predicate

- Generalizing to multiple operands is simple. However, the same limitations as in set-based implementations apply, and here it becomes clearer why supporting queries like "cat OR NOT(dog)" is not ideal. In our implementation, for the NOT(dog) operand, we would need to list all documents except those in dog's postings. Since document frequencies of terms can be low, enumerating NOT(dog) could involve millions or billions of document IDs, substantially slowing retrieval. On the other side, queries like "cat OR NOT(dog)" are not intuitive.
- We can use the same method for any mix of AND and OR operators nested within one another, as each evaluation method mentioned above produces sorted document IDs. Similar to single term searches, we can handle NOT operators when they are within an AND expression that contains at least one sub-expression without a NOT at the highest level (a nested NOT further down in the sub-expression is not an issue).

* We omit here a detailed discussion for the Extended Boolean Retrieval model. The approach is similar with the models to follow, that is, we first fetch all candidate documents (union of postings over all query terms) and then evaluate foe each document the overall score using one of the score combining functions.
* You can find a more detailed implementation here: https://github.com/roger-weber/mmir-unibasel/blob/main/chapter03/Boolean_InvertedIndex.ipynb

3.4.2 Inverted Files for the BIR model

- The Binary Independence Retrieval (BIR) model, Vector Space retrieval, Extended Boolean retrieval, and BM25 models exhibit several similarities when evaluated using inverted indexes. Conceptually, they adopt a retriever-ranker approach as previously explained:



  By utilizing inverted files, the retriever component retrieves the union of postings for the query terms. This yields a candidate list for the filter & ranker, which then employs the model's designated scoring function for each candidate document to generate the ranked list.

- Implementations frequently combine retriever/filter/ranker components for enhanced performance. We initially study the fundamental versions: **document-at-a-time** and **term-at-a-time** using the BIR model, owing to its uncomplicated scoring function (sum of c_i). Subsequently, we expand this to the vector space and BM25 models. The Extended Boolean model is omitted due to its diminished relevance in today's search contexts.
  - The **document-at-a-time** method retrieves documents consecutively through streaming like for the Boolean OR-operand approach. At each step, we obtain the document with the smallest doc ID from the sorted postings of each query term, and pass it along with its query terms to the scoring function. The ranker maintains a list of the best k documents encountered and maintains this list upon processing all candidates. The "top-k" mechanism minimizes storage needs, but still enables users to browse through several pages.
  - The **term-at-a-time** method goes through query terms one after the other. For each term, it updates the document list and uses the scoring function to adjust their scores based on that term's presence. At the end, documents are arranged by their overall score, forming the ranked list. Unlike document-at-a-time, this method cannot maintain a top-k list to reduce storage. However, it might suffer from long candidate lists if common terms with long postings are in the query. An optimization is to skip frequent terms during evaluation that are unlikely to change the ranking in a significant way.

- The Python code on the right shows a simplified version for the **document-at-a-time** retrieval technique for the BIR model. The `search_DAAT` function takes a query string, a desired number of results (`k`), and feedback data collected on documents.
  - We start by turning the query string into a set of words using a provided analyzer
  - Using feedback, we compute c_j-weights and trim terms. For instance, we might keep only the top-n weights from a larger set of query terms
  - The primary loop resembles the Or-implementation of the Boolean model. We sort the postings of each query term by document IDs. We iterate through the postings (`index[term]`) in a stream based manner (`iters`), selecting the smallest ID across the next elements (`nexts`) in the stream as a new candidate document id
  - If we have user feedback, we can skip 'non-relevant' documents. Otherwise, if the document is relevant or there's no feedback, we calculate the `score` by summing c_j-values (`term_weights[j][1]`), pairing it with the document's smallest ID, and adding it to the `topk` object. This object uses a heap to maintain `(doc_id, score)` tuples, ordered by `score` for efficient access to top-k results (no need for explicit sorting needed)
  - In the main loop's final step, we fetch the subsequent postings for each term where the smallest ID was at the stream's front (`nexts`)

```python
def search_DAAT(query, k, feedback):
    query_vector = analyzer.set_of_words(query)

    # filter terms and obtain c_j-weights
    term_weights = query_weights(query_vector, feedback)

    # get iterators for each term and fetch first posting
    iters = [iter(index[term]) for (term, _) in term_weights]
    nexts = [next(iter, None) for iter in iters]

    # keep track of all retrieved documents
    topk = TopKList(k)

    while not all(e is None for e in nexts):
        # get smallest value from nexts, ignoring None
        smallest = min(nexts, key = lambda x: x or math.inf)

        # use feedback, omit if assessed and not relevant
        if not feedback.is_assessed(smallest) or \
           feedback.is_relevant(smallest):
            # get score and add it to topk
            score = 0
            for j in range(len(nexts)):
                if nexts[j] == smallest:
                    score += term_weights[j][1]
            topk.add(smallest, score)

        # fetch next items if entry equals smallest
        for i, e in enumerate(nexts):
            if e is smallest:
                nexts[i] = next(iters[i], None)

    # finished, return topk for result iteration
    return topk
```

- Now, let's explore the **term-at-a-time** approach for the BIR model on the right side. The `search_TAAT` function takes a query string, a desired number of results (`k`), and feedback data collected on documents.
  - We start by turning the query string into a set of words using a provided analyzer.
  - Using feedback, we compute c_j-weights and trim terms. For instance, we might keep only the top-n weights from a larger set of query terms
  - The main loop runs through each query term (sorted by their weights in `query_weights`) and all postings (`index[term]`). It keeps track of a score for each seen document (dictionary `scores`)
  - If we have user feedback, we can skip 'non-relevant' documents. Otherwise, if the document is relevant or there's no feedback, we add the c_j-value of the current term (`weight`) to the scores dictionary. The update line also establishes new entries for previously unseen documents
  - Once the main loop concludes, the `scores` dictionary contains a value for each document that has at least one query term. Instead of directly sorting `scores`, we take a similar approach as with DAAT. We utilize the `TopKList` and include all document IDs and their corresponding scores
- You can find a more detailed implementation for both variants here: https://github.com/roger-weber/mmir-unibasel/blob/main/chapter03/BIR_InvertedIndex.ipynb

```python
def search_TAAT(query, k, feedback):
    query_vector = analyzer.set_of_words(query)

    # filter terms and obtain c_j-weights
    term_weights = query_weights(query_vector, feedback)
    scores = defaultdict(int)

    # iterate over terms and fetch postings
    for (term, weight) in term_weights:
        for doc_id in index[term]:
            # use feedback, omit if assessed and not relevant
            if feedback.is_assessed(doc_id) and \
               not feedback.is_relevant(doc_id):
                continue
            # add weight to score of document
            scores[doc_id] += weight

    # avoid full sort and use the heap in TopKList
    topk = TopKList(k)
    for doc_id, score in scores.items():
        topk.add(doc_id, score)

    # finished, return topk for result iteration
    return topk
```

- **Discussion: DAAT vs. TAAT**
  - Both methods have similar complexity in terms of the number of read postings. They both focus on documents that have at least one query term and a non-zero score
  - Both approaches can efficiently filter out previously marked non-relevant documents to prevent their reappearance in future results
  - The TAAT implementation is shorter and more concise but has a drawback—the `scores` dictionary. If query term postings are lengthy, this dictionary can become sizable
  - On the other hand, the DAAT approach computes scores in a single step for documents and adds them to a heap within the `TopKList` object. This heap not only provides efficient access in sorted order but can also be pruned occasionally if it becomes too large
- **Including Predicates in Evaluation:** We can expand both methods to search for documents with predicates like "star wars" and "year < 2000". The assessment of these queries depends on how we can evaluate the condition:
  1. Document attributes (metadata) in the predicate are stored in an index with an efficient evaluation plan. For instance, with the condition "year < 2000," we can use index lookup to find document IDs meeting the predicate. This index might be a B-tree or an inverted list
     - The optimal approach for text retrieval and predicate assessment is to first obtain all document IDs satisfying the predicate and then feed this selection (as an inverted list) to the search function
     - Inside the search function, we remove all candidates not included in the predicate selection. In the code, this adjustment aligns with where we check for non-relevant documents in the feedback
     - Apart from predicate evaluation, there is no additional complexity in the search algorithm
  2. If there is no index support for the condition, or the evaluation requires a full scan through all document data:
     - Since calculating the subset of documents satisfying the predicate is not efficient, we must assess the predicate individually when we return (in Python `yield`) results using the `TopKList` object
     - The heap within `TopKList` produces a stream sorted by decreasing score. Before delivering the object to the caller, we inspect the document's predicate (accessing metadata randomly). If the predicate is not met, we skip the document and move to the next one from the heap
     - In the best case (a less selective predicate), we evaluate the predicate for all documents returned as results, and a few omitted by the predicate. In the worst case (a highly selective predicate), we have to assess the predicate for all documents in the heap (still better than evaluating it over all documents)

3.4.3 Inverted Files for the Vector Space model

- In terms of the algorithms, both the BIR model and the Vector Space model are conceptually the same. The DAAT and TAAT implementations work similarly with these modifications:
  - Postings now comprise tuples with document IDs and term frequencies, sorted by document ID
  - Queries change into a bag-of-words model, including terms and their frequencies for the query
  - We need access to a vocabulary containing document frequencies. As an optimization, we can save required idf-weights alongside postings in the inverted files (to avoid random vocabulary accesses)
  - A similarity function that calculates scores based on the query vector and a document vector subset including query terms and their frequencies.
  - For cosine similarity, we additionally require the document vector's length (=$\|\boldsymbol{d}\|$)
  - For BM25, we also need the document length (number of term occurrences $|D|$), an average document length (adl), and parameters k and b for the calculation
- The inner vector product can compute all scores using the data in the inverted files (`index` in the implementation), but both the cosine measure and the BM25 similarity function need an extra lookup for document-related data (document length, norm of document vector). This can notably raise retrieval costs, demanding extra optimizations for consistent performance. To prevent such lookups, we can normalize document vectors at index build time.

$$sim_{cosine}(Q,D) = \sum_{j=1}^{M} \hat{d}_j \cdot \hat{q}_j \quad \text{with} \quad \hat{d}_j = \frac{\mathrm{idf}(\mathrm{t_j}) \cdot tf(D,t_j)}{\|\boldsymbol{d}\|} \quad \text{and} \quad \hat{q}_j = \frac{\mathrm{idf}(\mathrm{t_j}) \cdot tf(Q,t_j)}{\|\boldsymbol{q}\|}$$

$$sim_{BM25}(Q,D) = \sum_{j=1}^{M} idf(t_j) \cdot \hat{d}_j = \quad \text{with} \quad \hat{d}_j = \frac{tf(D,t_j) \cdot (k+1)}{tf(D,t_j) + k \cdot \left(1 - b + b\frac{|D|}{adl}\right)} \quad \text{and} \quad \mathrm{idf}(\mathrm{t_j}) = \log\left(\frac{N - df(t_j) + 0.5}{df(t_j) + 0.5}\right)$$

  If the normalization parameters (idf, k, b, $|D|$, adl) changes then we need to rebuild the index. Setting $q_j = idf(t_j)$ for the BM25, all three measures reduce to a dot-product between normalized document and query vector

- We omit here the code but you can find a detailed implementation here: https://github.com/roger-weber/mmir-unibasel/blob/main/chapter03/VectorSpace_InvertedIndex.ipynb

3.4.4 Inverted Files Implementation with SQL

- We can build traditional text retrieval using a database with inverted lists, created through database index structures. The code on the right outlines the steps for carrying out Boolean and vector space retrieval.

1. We generate tables for `documents`, `vocabulary`, and `postings`, along with a temporary table for the `query` of a search. The last setup creates an index over the `posting` table and `terms`. This builds a B-tree structure with document IDs and term frequencies in leaf nodes for swift retrieval in subsequent searches
2. Before re-building the collection, we delete all data from all tables
3. Next, we go through each document in the collection. For each document, we add an entry to the `document` table, form a bag-of-words representation of the document, and insert tuples `(term, docId, tf)` into the `posting` table.
4. We count the number of documents for the calculation of idf-weights. In the code on the right, we employ the standard formula, although we could choose any variant that fits the search scenario (for Boolean searches, `idf` and `tf` values are not used). Lastly, we count the document frequency and calculate idf-weights for each term by grouping the `posting` table by `terms` and inserting the outcomes into the `vocabulary` table.

```sql
-- 1. create schema for inverted index
--    document table can have additional attributes
--    auto incremented doc IDs depends on database product
CREATE TABLE document(id SERIAL PRIMARY KEY,
                      title TEXT, year INTEGER)
CREATE TABLE vocabulary(term TEXT, df INTEGER, idf REAL)
CREATE TABLE posting(term TEXT, docId INTEGER, tf INTEGER)
CREATE TEMPORARY TABLE query(term TEXT, tf INTEGER)
CREATE INDEX inverted_list ON posting(term)


-- 2. rebuild index from documents
--    delete all existing data
DELETE FROM posting
DELETE FROM vocabulary
DELETE FROM document


-- 3. for all documents in collection (outside of database)
--    fetch id after next insert (database dependent)
INSERT INTO document(title, year) VALUES (:title, :year)

--    create a bag-of-word representation and insert
INSERT INTO posting(term, docId, tf) VALUES(:term, :id, :tf)


-- 4. build vocabulary (table vocabulary)
--    fetch number of documents --> ndocs
SELECT count(*) AS count FROM document

--    insert terms from posting table into vocabulary table
INSERT INTO vocabulary(term, df, idf)
      SELECT term,
             count(*),
             ln(1.0 * (:ndocs + 1) / (count(*) + 1))
        FROM posting
    GROUP BY term
```

5. For Boolean AND-searches with 2 terms, we join the `posting` table with itself and pick postings for search terms (`:term1`, `:term2`) sharing the same `docId`. Since we created an index over `posting(term)`, the database will execute two B-tree lookups to retrieve lists of `(docId, tf)` from leaf nodes and matching them with the `WHERE`-clause. Finally, we join the results with the `document` table to provide document details. A Boolean OR-search does not require a self-join of the `posting` table, yet the query still involves 2 B-tree lookups, matching the `WHERE`-clause, and merging with the `document` table to return results. While OR-queries might seem simpler (fewer joins), their evaluation complexity is the same.
6. To handle any number of query terms, we utilize a temporary `query` table and populate it with query terms (using `tf=1` following the set-of-words model). For AND-queries, we link the `posting` and `query` table. The database executes a B-tree lookup for each query term, grouping them by `docId`. When a `docId`-group contains as many entries as there are query terms, it satisfies the AND-condition. We then combine these results with the `document` table to create the response. For OR-queries, we apply the same process, but we omit the `HAVING`-clause since we return all documents having at least one matching query term. Query evaluation complexity grows linearly with the number of query terms.

```sql
-- 5. boolean query with 2 terms
--    :term1 AND :term2
SELECT d.*
  FROM document d, posting a, posting b
 WHERE a.term = :term1 AND
       b.term = :term2 AND
       a.docId = b.docId AND
       a.docId = d.id

--    :term1 OR :term2
SELECT d.*
  FROM document d, posting a
 WHERE a.term IN (:term1, :term2) AND
       a.docId = d.id



-- 6. boolean query with arbitrary number of terms
--    add query terms to temporary table
DELETE FROM query
INSERT INTO query(term, tf) VALUES(:term1, 1)
INSERT INTO query(term, tf) VALUES(:term2, 1)
…

--    AND(:term1, :term2, ...)
  SELECT d.*
    FROM document d, posting p, query q
   WHERE p.term = q.term AND
         p.docId = d.id
GROUP BY p.docId
  HAVING COUNT(p.term) = (SELECT COUNT(*) FROM query)

--    OR(:term1, :term2, ...)
  SELECT d.*
    FROM document d, posting p, query q
   WHERE p.term = q.term AND
         p.docId = d.id
GROUP BY p.docId
```

7. Using the temporary `query` table, we can implement various vector space models. In the code on the right, we provide an example using the dot-product measure. Similar to before, we insert the query terms into the `query` table and then join the `query` table with the `posting` table. This triggers B-tree lookups for the `posting` table for each query term, and we group the postings by `docId`. Since vector space models function like an OR-Boolean query for candidate selection, a `HAVING`-clause is not required. However, we need to join the results with both the `document` and `vocabulary` tables to calculate the scores. The final `ORDER BY` clause arranges the documents by decreasing scores.
8. Consider how we integrated predicates in the Python retrieval implementation. We either start by fetching the set of `docId`s fulfilling the predicate and include that selection in the retrieval process, or we must create a result stream from the retrieval system and then individually filter documents through database-based predicate evaluation. The latter approach, particularly, is inefficient (while the former might not be supported by the retrieval system). If we unify predicate evaluation and text retrieval within the database, as demonstrated on the right for vector space retrieval and the "year > 1990" predicate, we get faster and simpler evaluation plans.

- You can find a more detailed implementation here: https://github.com/roger-weber/mmir-unibasel/blob/main/chapter03/SQL_InvertedIndex.ipynb

```sql
-- 7. vector space model with dot product
--    add query terms to temporary table
DELETE FROM query
INSERT INTO query(term, tf) VALUES(:term1,:tf1)
INSERT INTO query(term, tf) VALUES(:term2,:tf2)
…

--    calculate dot product and order by score
   SELECT SUM(p.tf * v.idf * q.tf * v.idf) AS score, d.*
     FROM document d, posting p, query q, vocabulary v
     WHERE p.term = q.term AND
           p.term = v.term AND
           p.docId = d.id
GROUP BY p.docId
ORDER BY 1 DESC
```

```sql
-- 8. adding predicates (example with vector space model)
--    add predicates on attributes in document table
  SELECT SUM(p.tf * v.idf * q.tf * v.idf) AS score, d.*
    FROM document d, posting p, query q, vocabulary v
   WHERE p.term = q.term AND
         p.term = v.term AND
         p.docId = d.id AND
         d.year > 1990
GROUP BY p.docId
ORDER BY 1 DESC
```

3.5 Lucene - Open Source Text Search

- **Lucene**, initiated by Doug Cutting in 1997, is an open-source Java-based information retrieval software. Its goal was to offer modern text analytics, indexing, and search functions. In 2000, Lucene's initial stable version (v1) was launched, and within a year, it became an **Apache Software Foundation** project. From its inception, Lucene has served as a prominent basis for operational search applications and has received consistent updates over time.
  - Releases: 2000 (v1), 2006 (v2), 2009 (v3), 2012 (v4), 2015 (v5), 2016 (v6), 2017 (v7), 2019 (v8), 2021 (v9)
  - Major releases may coincide, but Lucene typically discontinues development and support for older releases. Nevertheless, you can choose to use older versions, but it comes with risks. Lucene employs major releases to update APIs, enhance interaction with engine components, and improve integration of third-party extensions. As a results, you may frequently find examples on the web that no longer compile with the latest version.
  - This chapter works with **Lucene v9.7.0** and examples may not work with older/newer versions.
- Lucene is incorporated into a variety of products across different domains. Here are some notable products and applications that use Lucene:
  - **Elasticsearch**, a popular search and analytics engine built on Lucene, is renowned for full-text search, log analysis, and other search-based applications. The ELK stack (Elasticsearch, Logstash, Kibana) is widely employed for log analytics and SIEM (security information and event management). In 2021, Elastic's licensing change led to the emergence of **OpenSearch** as an Apache 2.0 open-source fork.
  - **Apache Solr**, also based on Lucene, is a widely-used search platform with features such as faceted search, distributed search, and advanced text analysis. It is employed by Cloudera, DataStax, Bloomberg, eBay, Netflix, and Amazon CloudSearch to drive enterprise search engines.
  - Apache **OpenNLP**, used for natural language processing tasks, uses Lucene for document indexing and search.
  - Many products by Atlassian, including **Jira**, **Confluence**, and **Bitbucket**, use Lucene for their search functionality.
  - The Hadoop ecosystem (**Apache Pig, Apache Hive**) can use Lucene for full-text search and indexing.
  - **Wikipedia**'s search functionality is powered by Lucene. It allows users to search for articles and find relevant content. First, Wikipedia used Lucene directly, after 2014 via Elasticsearch.
  - **Apache Cassandra**, a distributed NoSQL database, has integration with Lucene for full-text search capabilities.
  - An (incomplete) list can be found here: https://cwiki.apache.org/confluence/display/lucene/PoweredBy

- Lucene is split into a number of packages: lucene-core is the main package and provides the fundamental analysis, indexing, and search capabilities. It is extended by (see Apache Lucene home page for a complete list):
  - lucene-analysis-common: Analyzers for indexing content in different languages and domains
  - lucene-queryparser: Query parsers and parsing framework
  - lucene-analysis-opennlp: OpenNLP Library Integration
  - lucene-analysis-phonetic: Analyzer for indexing phonetic signatures (for sounds-alike search)
  - lucene-benchmark: Lucene benchmarking module
  - lucene-classification: Classification module for Lucene
  - lucene-facet: Faceted indexing and search capabilities
  - lucene-queries: Filters and Queries that add to core Lucene
  - lucene-suggest: Auto-suggest and Spellchecking support
- Each package offers additional features and must be included alongside lucene-core. Depending on your build tool, specify the group (org.apache.lucene), name (lucene-core), and version (9.7.0) to retrieve the libraries from a Maven repository (for example: https://mvnrepository.com/artifact/org.apache.lucene)
  - Example in Gradle (build.gradle):

```
dependencies {
    implementation group: 'org.apache.lucene', name: 'lucene-core’,            version: '9.7.0'
    implementation group: 'org.apache.lucene', name: 'lucene-analysis-common’, version: '9.7.0'
    implementation group: 'org.apache.lucene', name: 'lucene-queryparser’,     version: '9.7.0'
}
```

  - Example in Maven (pom.xml)

```
<dependency>
    <groupId>org.apache.lucene</groupId>  <artifactId>lucene-core</artifactId>              <version>9.7.0</version>
</dependency>
<dependency>
    <groupId>org.apache.lucene</groupId>  <artifactId>lucene-analyzers-common</artifactId>  <version>9.7.0</version>
</dependency>
<dependency>
    <groupId>org.apache.lucene</groupId>  <artifactId>lucene-queryparser</artifactId>       <version>9.7.0</version>
</dependency>
```

- The core model of Lucene considers a document as a set of fields whereby not each document must have each field. Fields can be of different types, can be tokenized, and can be stored in Lucene's data structure.
  - The `Field` base class accepts a name, a value, and a `FieldType` that specifies tokenization, indexing, and storage settings. When a field is tokenized, its value can be used for full-text search. Index options determine what information is stored in the inverted index, including document IDs, term frequencies, and positions in the value. If a field is stored, search results provide interfaces to access its value. If it is not stored, the application must retrieve the value from its own database. Importantly, tokenization and indexing are independent of storage. Even if a field is tokenized but not stored, full-text searches can still be performed on it. This is actually the norm for longer fields such as the document body field with the bulk of text data.
  - Different Field subclasses define field types for common scenarios:
    - `TextField`: Indexes and tokenizes a field (optional store) with document IDs, term frequencies, and positions
    - `StringField`, `IntField`, and `FloatField`: Used for document metadata. These fields are not tokenized but indexed and support exact or range queries. Storage can be enabled or disabled
    - `StoredField`: Only stores the value without indexing or tokenization (not searchable). Useful for internal document IDs or links to document locations
- When documents have differently named tokenized fields, the terms in these fields are treated independently. For example, if the "title" field has the term "house" and the "summary" field also has the term "house", these two occurrences are distinct and searches for "house" need to specify the applicable fields. Internally, Lucene prefixes the terms in the "title" and "summary" fields such that the "house" occurrences are actually seen as "title:house" and "summary:house". This is a powerful concept to treat fields differently during retrieval, and to use distinct normalization methods for each field. As an example, we can weigh title matches higher than full text matches.
- Lucene creates an inverted index, including a dictionary and postings, while storing document fields based on field type specifications. It offers various storage formats, notably a compound file-based index structure. Lucene, however, divides the index into smaller immutable segments, created each time an `IndexWriter` is opened for document addition. This approach reduces concurrency issues and safeguards against segment corruption. To merge smaller segments into larger ones, you can configure a `MergePolicy`. Due to segment immutability, delete and update operations require an additional file to mark documents as deleted. In an update, Lucene first marks the current document as deleted before adding a new one. Documents are never deleted in segments, just marked as deleted. When a merge occurs, deleted documents are removed.

To study Lucene's analyzer, we use a JShell session on the right side (more details here: https://github.com/roger-weber/mmir-unibasel/blob/main/chapter03/lucene.ipynb)

- The `print_tokens` function opens a token stream for a field named "text" with the given analyzer and text as arguments. `TokenStream` uses a visitor pattern to enumerate data aspects. We utilize `CharTermAttribute` to print tokens.

1. The `StandardAnalyzer` eliminates punctuation (excluding possessive-'), converts tokens to lowercase, and optionally filters out stop words (if given).
2. The `EnglishAnalyzer`, on top, removes possessive forms and uses a Porter stemmer after excluding the top-33 English words.
3. We can modify the analyzers by providing a custom stop word list. For instance, with an `EnglishAnalyzer`, we can remove all instances of 'i' and 'do'.
4. Lastly, we can create custom analyzers. In this example, we remove English possessive forms, filter out terms with fewer than 4 characters, and employ a dictionary-based stemmer while retaining term casing.

Once selected, we have to use the same analyzer for all documents and queries or we need to rebuild the index

```
void print_tokens(Analyzer analyzer, String text) throws IOException {
  TokenStream ts = analyzer.tokenStream("text", new StringReader(text));
  CharTermAttribute termAtt = ts.addAttribute(CharTermAttribute.class);

  for(ts.reset(); ts.incrementToken();)
    System.out.print(termAtt.toString() + " ");
  ts.end();
  System.out.println();
}

var text = "I think text's values' color goes here; WHAT happens with …";
var stopWords = new CharArraySet(Arrays.asList("i", "do"), false);

// 1. Standard analyzer
print_tokens(new StandardAnalyzer(), text);
↳ i think text's values color goes here what happens with it do we see …

// 2. English analyzer (from lucene-analysis-common)
print_tokens(new EnglishAnalyzer(), text);
↳ i think text valu color goe here what happen do we see again i went …

// 3. English analyzer with stopwords 'i' and 'do'
print_tokens(new EnglishAnalyzer(stopWords), text);
↳ think text valu color goe here what happen with it we see it again …

// 4. Custom analyzer filtering out short words
class MyAnalyzer extends Analyzer {
  protected TokenStreamComponents createComponents(String fieldName) {
    Tokenizer source = new StandardTokenizer();
    TokenStream result = new EnglishPossessiveFilter(source);
    result = new FilteringTokenFilter(result) {
      private CharTermAttribute ta = addAttribute(CharTermAttribute.class);
      protected boolean accept() throws IOException {
        return ta.length() > 3;
      }
    };
    return new TokenStreamComponents(source, new KStemFilter(result));
   }
}

print_tokens(new MyAnalyzer(), text);
↳ think text value color go here WHAT happen with again went there …
```

The code on the right demonstrates hot to build an index with Lucene:

1. First, we define an analyzer as per previous page. Next, we decide how to store the index (`Directory`). Lucene offers various options, with the file system directory being the most common choice. Finally, we create an `IndexWriter` object with the chosen analyzer and directory (Lucene offers additional configuration options). This `IndexWriter` allows us to add and modify documents in the index.
2. As previously discussed, documents are constructed as sets of fields. For each field, we can utilize higher-level classes like `TextField` or `IntField` and define whether to tokenize, index, and store the fields. In the example on the right, we tokenize both the `title` and `body`, index all fields, and store `title` and `year`. Thus, we cannot access `body`-values in result objects.
3. When loading data into an index, we create an `IndexWriter` and add documents with it. Each new `IndexWriter` generates a new segment. If we specify a merge policy (enabled by default), `IndexWriter`s will also merge segments as needed. In the example on the right, data is loaded in batches of 100 documents, and each batch results in the creation of a new segment.

```
// 1. define analyzer, directory, and index writer
Analyzer getAnalyzer() {
   return new EnglishAnalyzer();
}

Directory getDirectory() throws IOException {
   return FSDirectory.open(Paths.get(“./index"));
}

IndexWriter getIndexWriter() throws IOException {
   Directory directory = getDirectory();
   IndexWriterConfig config = new IndexWriterConfig(getAnalyzer());
   return new IndexWriter(directory, config);
}

// 2. build a document from key-value data
Document createDocument(Map<String, String> data) {
   Document doc = new Document();
   doc.add(new TextField("title", data.get("title"), Store.YES));
   doc.add(new IntField("year", Integer.parseInt(data.get("year")), Store.YES));
   doc.add(new TextField("body", data.get("body"), Store.NO));
   return doc;
}

// 3. load data in batches
void loadBatch(List<Map<String, String>> docs) throws IOException {
   IndexWriter writer = null;

   writer = getIndexWriter(false);
   for(Map<String, String> doc : docs)
      writer.addDocument(createDocument(doc));
   writer.close();     // ensure that we close the index (better use finally)
}

void loadData(int batchSize) throws IOException {
   List<Map<String, String>> collection = readCollection();

   for(int i = 0; i < collection.size(); i += batchSize)
      loadBatch(collection.subList(i, Math.min(i + batchSize, collection.size())));
}

// load collection in batches; each batch creates a new segment
loadData(100);
```

Searching with Lucene is shown on the right:

1. For query execution, we require an `IndexSearcher` for the chosen `Directory` type. Additionally, we employ a query parser to enable users to submit full-text queries. The `MultiFieldQueryParser` generates queries from text against all tokenized fields. Lucene would typically require term queries against a single field but this helper class makes it simple to search against `title` and body `field` at the same time. The query parser needs the exact same analyzer as we used for indexing documents.
2. Queries yield a `TopDocs` object that enables sub-query merging (segments, shards) and contains the `scoreDocs` attribute, providing access to the best-matching documents and their scores. To print the `title` and `year` (the `body` field was not stored), we can retrieve the document by its ID (`doc.doc`) and access the fields using the `get`-method.
3. Lastly, we offer a straightforward helper method that takes a query and conducts a search using the `IndexSearcher` object created in step 1. To keep the code simple, we always retrieve the top 10 matches for each submitted query. In a practical implementation, we could provide an extra parameter to specify the number of results.

```
// 1. define analyzer, directory, index searcher, and a query parser
IndexSearcher getIndexSearcher() throws IOException {
   return new IndexSearcher(DirectoryReader.open(getDirectory()));
}

QueryParser getQueryParser() throws IOException {
   return new MultiFieldQueryParser(new String[]{"title", “body"}, getAnalyzer());
}



// 2. print results with values stored in index
void printResults(TopDocs results) throws IOException {
   int rank = 1;
   System.out.printf("%3s %5s %6s %6s   %s\n",
                                "#", "id", "Score", "Year", "Title");
   for(ScoreDoc doc: results.scoreDocs) {
      Document document = getIndexSearcher().doc(doc.doc);
      System.out.printf("%3d %5d %6.2f %6s   %s\n",
                                rank++, doc.doc, doc.score,
                                document.get("year"), document.get("title"));
   }
}



// 3. executing a query and printing the top-10 matches
void searchQuery(Query query) throws IOException {
   printResults(query.toString(), getIndexSearcher().search(query, 10));
}
```

Finally, we can build queries, evaluate them, and show their results:

1. The examples on the right use the class `MultiFieldQueryParser`. This parser offers an intuitive syntax for queries:
   - Keyword lists execute BM25 searches against both the `title` and `body` fields (unless a different similarity measure is defined).
   - `title:star^2` narrows the search to the `title` field with a weight of 2. This grants greater control over term occurrence and importance.
   - `title:{a TO b}` specifies a range search for terms between `a` and `b`, essentially performing a search with all terms that start with a.
   - `fuzzy~0.6` conducts a search for terms closely related (0.6 defines the closeness) to "fuzzy". This is useful for finding misspelled terms or valid variants.
   - `z?rich` enables wildcard matches against keywords and searches for all matching terms.
2. The second example demonstrates how to construct custom queries. Using the `BooleanQuery.Builder` helper, we can add new query components, including nested queries, such as field predicates (e.g., `year == 2020`), term searches, and various other types like those used in the examples above. For each query component, we can specify its impact: as a predicate that does not affect scoring (`Occur.FILTER`) or one that should, must, or must not occur and influences scoring (`Occur.MUST`, `.SHOULD`, `.MUST_NOT`).

```
// 1. using multi-field query parser
searchQuery(getQueryParser().parse("star wars"))
searchQuery(getQueryParser().parse("title:star title:wars"))
searchQuery(getQueryParser().parse("title:star^2 title:wars"))
searchQuery(getQueryParser().parse("title:{a TO b}"))
searchQuery(getQueryParser().parse("fuzzy~0.6"))
searchQuery(getQueryParser().parse("z?rich"))
```

```
// 2. using Boolean query builder
searchQuery(new BooleanQuery.Builder()
      .add(IntField.newExactQuery("year", 2020), Occur.FILTER)
      .add(new TermQuery(new Term("title","stars")), Occur.SHOULD)
      .add(new TermQuery(new Term("title","wars")), Occur.SHOULD)
      .build()
)
// .add(IntField.newExactQuery("year", 2020), Occur.MUST)
// .add(IntField.newExactQuery("year", 2020), Occur.SHOULD)
// .add(IntField.newRangeQuery("year", 2010, 2020), Occur.FILTER)
```

A notable feature of Lucene is the `IndexSearch.explain()` function, which provides a comprehensive explanation of how a document's score was computed. Consider the example query on the right and the score calculations applied to the top result:

- Filter predicates (`qYear1`) must be satisfied but have no impact on scoring.
- `SHOULD` predicates (`qYear2`) contribute a score of `1.0` to the overall result when met.
- `MUST` term queries (and similarly `MUST_NOT`) have to be met and affect the score. The details reveal the use of the BM25 formula with certain adjustments:

$$idf = \log\left(1 + \frac{N-n+0.5}{n+0.5}\right)$$

$$score = boost \cdot idf \cdot \frac{tf \cdot (k_1 + 1)}{tf + k_1 \cdot \left(1 - b + b\frac{dl}{adl}\right)}$$

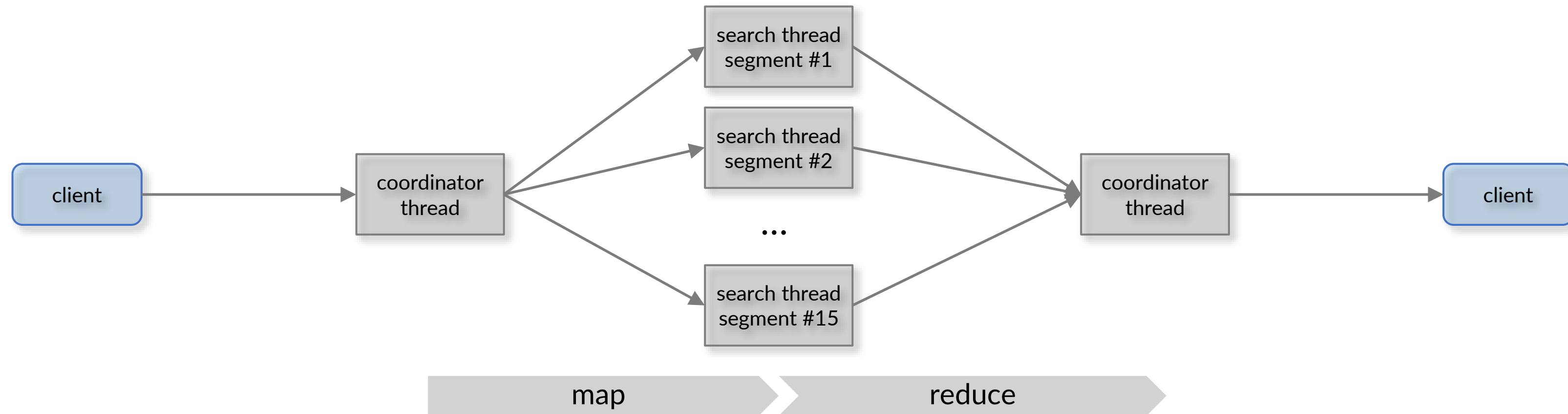
  - `n` represents the document frequency, calculated individually for each field.
  - `boost` is an additional factor for the query part (as seen in `qBody` with `boost = 1.5`).
  - `k1` and `b` are parameters for term frequency normalization in line with the BM25 formula. Document length (`dl`) and average document length (`adl`) pertain to a single field, not the entire document.
- The final score is the sum of all query parts.

```
qYear1   = IntField.newRangeQuery("year", 1950, 2000);
qYear2   = IntField.newRangeQuery("year", 1990, 2000);
qTitle   = new TermQuery(new Term("title", "shawshank"));
qBody    = new TermQuery(new Term("body", "decency"));

query = new BooleanQuery.Builder()
               .add(qYear1, Occur.FILTER).add(qYear2,Occur.SHOULD)
               .add(qTitle, Occur.MUST)
               .add(new BoostQuery(qBody, 1.5f),Occur.SHOULD)
               .build();

results = searcher.search(query, 10);

// explain results
System.out.println(searcher.explain(query, results.scoreDocs[0].doc));
↪
    7.4037647 = sum of:
      0.0 = match on required clause, product of:
        0.0 = # clause
        1.0 = year:[1950 TO 2000]
      1.0 = year:[1990 TO 2000]
      3.0980327 = weight(title:shawshank in 0) [BM25Similarity], result of:
        3.0980327 = score(freq=1.0), computed as boost * idf * tf from:
          6.5022902 = idf, computed as log(1 + (N - n + 0.5) / (n + 0.5)) from:
            1 = n, number of documents containing term
            999 = N, total number of documents with field
          0.47645253 = tf, computed as freq / (freq + k1 * (1 - b + b * dl / avgdl)) from:
            1.0 = freq, occurrences of term within document
            1.2 = k1, term saturation parameter
            0.75 = b, length normalization parameter
            2.0 = dl, length of field
            2.2532532 = avgdl, average length of field
      3.3057323 = weight(actors:decency in 0) [BM25Similarity], result of:
        3.3057323 = score(freq=1.0), computed as boost * idf * tf from:
          1.5 = boost
          4.7686887 = idf, computed as log(1 + (N - n + 0.5) / (n + 0.5)) from:
            8 = n, number of documents containing term
            1000 = N, total number of documents with field
          0.46214414 = tf, computed as freq / (freq + k1 * (1 - b + b * dl / avgdl)) from:
            1.0 = freq, occurrences of term within document
            1.2 = k1, term saturation parameter
            0.75 = b, length normalization parameter
            8.0 = dl, length of field
            8.335 = avgdl, average length of field
```

3.5.5 Apache Solr, Elasticsearch, and OpenSearch

- Lucene scales effectively up to its maximum limit of 2.1 billion documents. Its segment-based architecture allows parallelization for individual searches. However, when the index expands beyond approximately 20-40GB, search times are constrained by the maximum I/O and/or memory throughput. Moreover, if we need to run hundreds of concurrent queries, the core structure of Lucene is not suitable for handling such loads.
- To address theses limitations, three popular options exist which are built on top of Lucene and provide powerful tools for distributing and scaling search operations:
  - **Apache Solr** is a highly flexible and extensible search platform, expanding upon Lucene's capabilities. It offers features like distributed searching, load balancing, and real-time indexing. Solr is user-friendly and capable of managing large-scale search tasks. Its centralized configuration simplifies management and scaling across server clusters. Solr also supports faceted search, allowing results to be grouped by facets such as category, country, or other user-defined dimensions. It is a popular choice as a search engine on major websites and is integrated into popular big data platforms.
  - **Elasticsearch** is renowned for its real-time distributed search and analytics capabilities. It is designed for horizontal scalability, focusing on distributed use cases. Elasticsearch is commonly employed in log analytics and security analytics scenarios. Alongside Logstash and Kibana, it forms the widely used ELK stack for observability applications. Elasticsearch builds upon Lucene's core features to enable field and text search. Although used extensively with logs, it features a contemporary full-text document search.
  - **OpenSearch**, initiated in 2021, is an Apache 2.0 fork of Elasticsearch. This move came in response to Elastic's decision to alter the licensing terms of Elasticsearch. The new dual licensing model affected not only cloud vendors but also smaller vendors and upset the open-source community. Led by Amazon, the OpenSearch community now offers an alternative solution that remains fully compatible with Elasticsearch.
- Solr, Elasticsearch, and OpenSearch utilize sharding as a fundamental scaling technique. Sharding divides the index into smaller, autonomous components known as shards, with each shard capable of residing on a separate cluster node. Moreover, each shard can be replicated multiple times within the cluster to enhance availability and scalability. Sharding accelerates individual queries, and shard replication boosts the capacity for concurrent queries. Importantly, sharding eliminates the 2.1 billion document constraint of Lucene.

- Let's study the different levels to scale out searches:
  1. At the **segment level**, Lucene can divide the index into multiple segments. You can specify a merge policy to control the number and size of these segments. To simplify, let's consider a merge policy that keeps 15 segments of equal size, and let's assume a server with 16 vCPUs. A coordinator thread (1 vCPU) handles query parsing and assigns the 15 segments to searcher threads running on the remaining 15 vCPUs. Each search thread conducts the search within its assigned segment, returns the result to the coordinator thread, which consolidates the results and sends the answer back to the client.

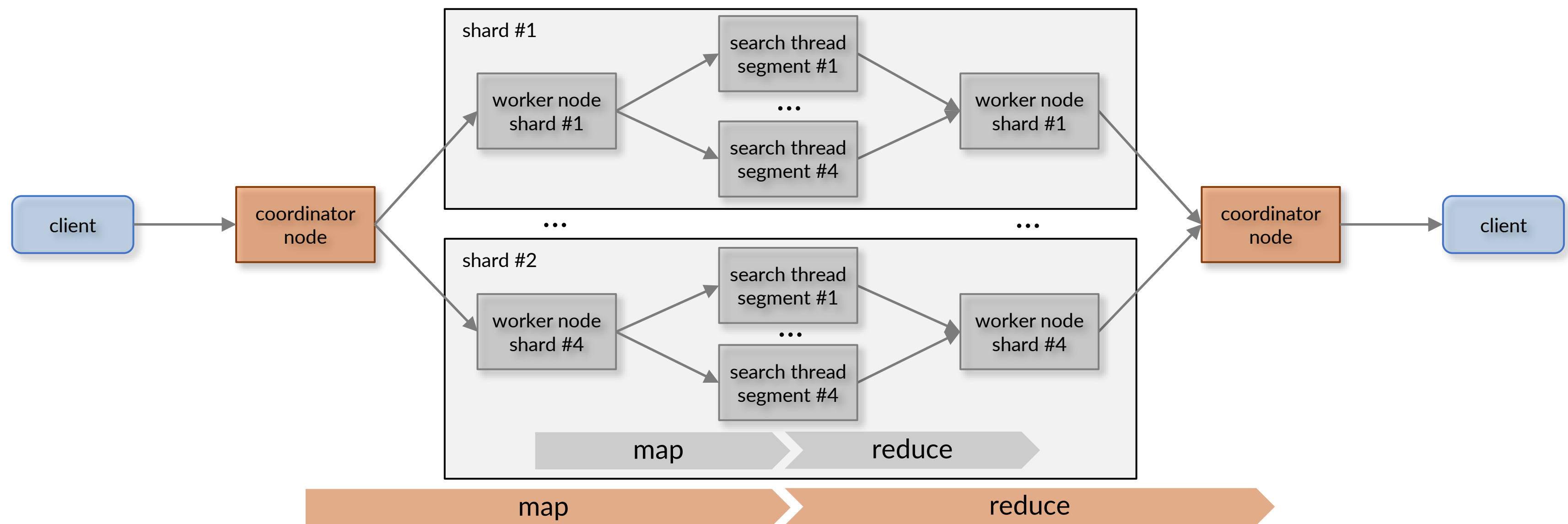


Applying Amdahl's law and assuming 99% of the time is spent on segment searching, we can achieve a 13-fold speedup compared to a single thread. Nevertheless, search times increase proportionally with the index size (until Lucene's limitations are reached), and we can only execute one query at a time or decrease the number of parallel threads. However, using larger machines (e.g., 128 vCPUs) is excessive and leads to a monolithic search server which lacks high availability.

2. To enhance scalability, Solr, Elasticsearch, and OpenSearch employ **sharding** to distribute collections across multiple smaller worker nodes. Each shard contains a unique subset of the documents, managed by separate Lucene indexes. Each index maintains its own segments and term statistics which can result in varying scores for identical documents stored in different shards. As search is inherently not an exact operation, some inconsistency in scores is acceptable as long as the deviations are not substantial across shard assignments.

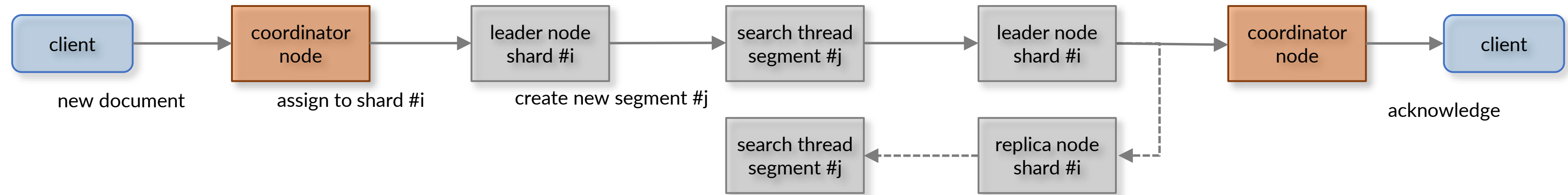
   When a new document is introduced to the collection, a coordinator node determines the shard to which it is assigned. This assignment can be based on predefined policies like round-robin, or by utilizing user-defined prefixes on the document ID which are hashed to a specific shard number (ensuring that documents with the same prefix go to the same shard). Typically, the number of shards remains constant since redistributing and reinserting documents is a resource-intensive operation. To increase parallelism, some tools may necessitate recreating the search domain and reinserting all documents to achieve an even distribution across the shards.

   During search operations, we now employ two levels of map-reduce: one at the shard level and, within each shard, at the segment level. Despite each shard having slightly different term statistics, we can efficiently merge result lists by considering the overall score of the documents.

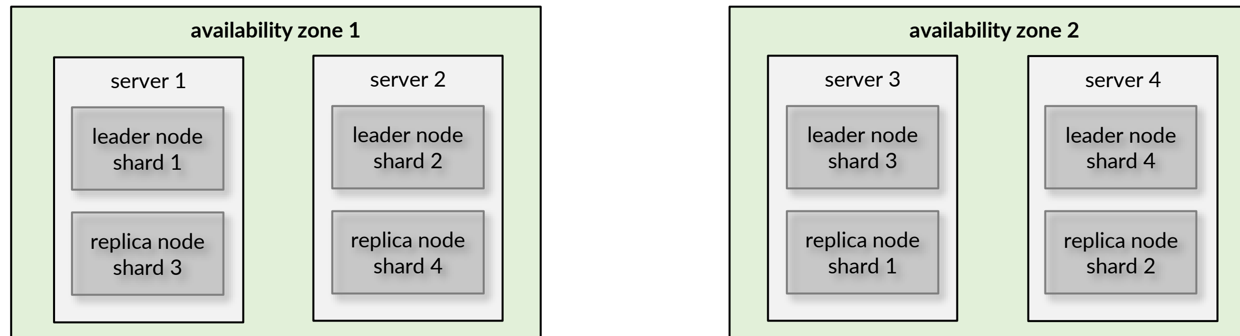


   With this configuration, we can lower our hardware demands and utilize numerous smaller worker nodes. These nodes can be operated within a Kubernetes cluster distributed across physical servers. Rapidly deploying new worker nodes and responding promptly to node failures enhances overall availability, as we will discuss next.

3. To improve availability, we **replicate shards** and distribute these replicas across different availability zones. Replicas of the same shard are intentionally placed on nodes located separately—avoiding assignment to nodes on the same server, within the same rack, or within the same data center. Apache Solr, Elasticsearch, and OpenSearch execute replication at the storage level to guarantee identical results from each replica. Within each shard, a leader node is responsible for adding documents to its index and subsequently disseminating updates to nodes that maintain replicas of the corresponding shard.

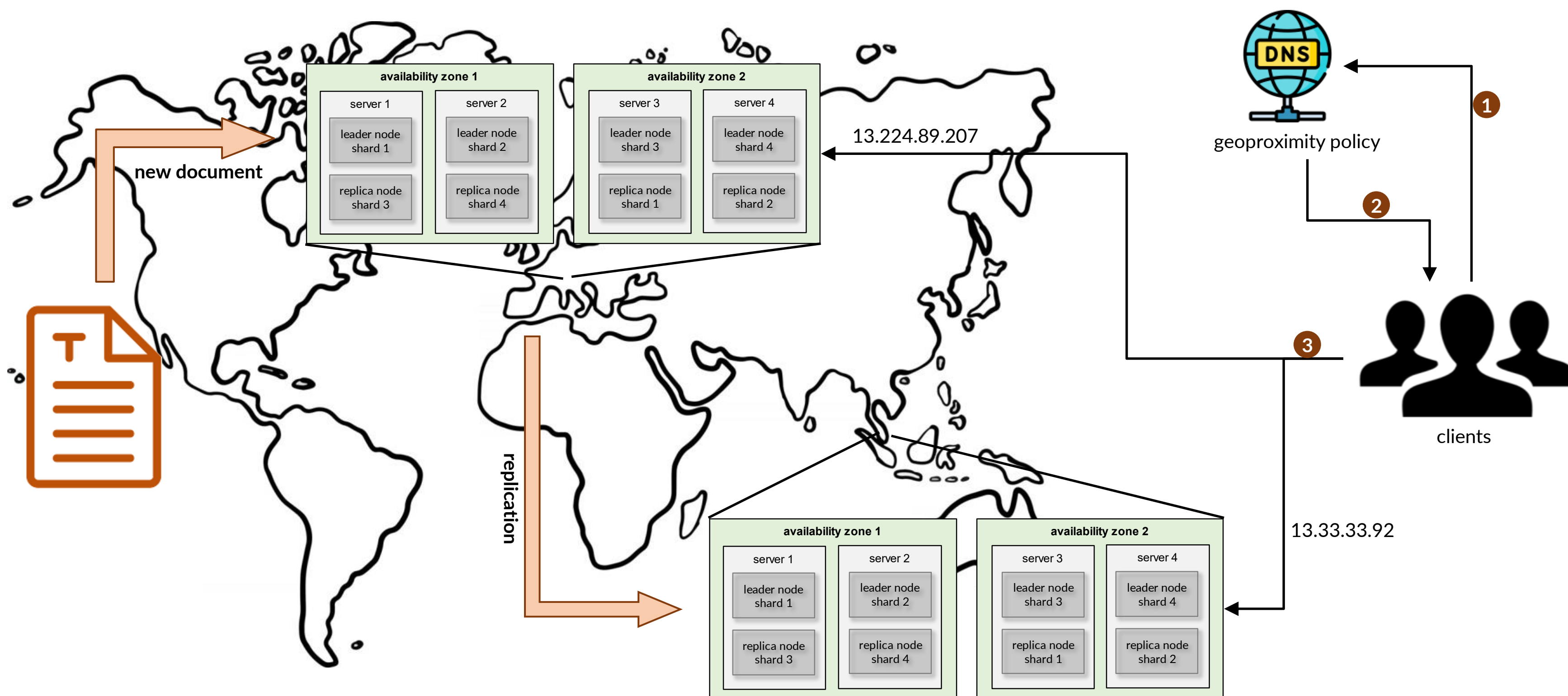


The number of replicas influences overall system availability and the system's resilience against failures. In our example with 4 shards and 2 replicas distributed across 2 availability zones, it might look as follows (each leader node can take over the role of the coordinator role as well):



Replicas not only boost availability but also enable more simultaneous searches. Each leader node can act as a coordinator, routing requests to a replica (or leader) node for each shard. As we increase the number of replicas, we can accommodate more concurrent searches. Because only leader nodes index documents and manage search coordination, we can scale concurrent searches in direct proportion to the number of replicas, up to the capacity of leader/coordinator nodes. To scale in accordance with the number of replicas, additional physical servers must be added to the cluster, although these servers can still host relatively small nodes (containers).

4. The highest level of distribution deploys instances across **multiple regions**, strategically placed near the target user population. For example, to serve users in both Europe and Asia, we can establish a search cluster in each region. Document insertions occur in a primary region, and shard updates are replicated to the other region. A DNS router equipped with a geoproximity policy routes client requests to the nearest region, with the alternative region serving as a backup. However, due to substantial distances and ping latencies of 100-200ms between Europe and Asia, we cannot distribute the execution of a single search across two regions to achieve further scalability. Nevertheless, we can enhance the number of concurrent searches, the overall availability of our search application, and reduce search latency for clients. Without this regional setup, clients in the other region would experience increased ping latencies.



(1) client requests DNS service to resolve hostname of search application (e.g., http://search.me/login)
(2) DNS service responds with IP address of the regional entry point closest to the client
(3) client requests service in region with given IP address (if this fails, tries the other region)

2.7 Literature and Links

General Books on Text Retrieval

- Gerard Salton, Michael J. McGill. **Introduction to Modern Information Retrieval**, McGraw-Hill Book Company, 1983.
- W.B. Frakes and R. Baeza-Yates. **Information Retrieval, Data Structures and Algorithms**, Prentice Hall, 1992.
- Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze, **Introduction to Information Retrieval**, Cambridge University Press. 2008. https://nlp.stanford.edu/IR-book/information-retrieval-book.html
- 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, 1998/2004.
- Ricardo Baeza-Yates and Berthier Ribeiro-Neto. **Modern Information Retrieval**, ACM Press Books, 1999/2011.
- Sandor Dominich. **Mathematical Foundations of Information Retrieval**, Kluwer Academic Publishers, 2001.
- S. Büttcher, C. Clarke, G. Cormack. **Information Retrieval - Implementing and Evaluating Search Engines**. MIT Press 2010.

Selected Articles on Retrieval models

- G. Salton, A. Wong, and C. S. Yang, **A Vector Space Model for Automatic Indexing,** Communications of the ACM, 1975 (Article in which a vector space model was presented) https://doi.org/10.1145/361219.361220
- Robertson, S. E.; Jones, K. Spärck, **Relevance weighting of search terms**. Journal of the American Society for Information Science, 1976. https://doi.org/10.1002/asi.4630270302
- Spärck Jones, K., **A Statistical Interpretation of Term Specificity and Its Application in Retrieval**. Journal of Documentation, 1972. https://doi.org/10.1108/eb026526
- Porter, M.F. (1980), **An algorithm for suffix stripping**, Program: electronic library and information systems, Vol. 14 No. 3. https://doi.org/10.1108/eb046814
- Wikipedia on Zipf's law, https://en.wikipedia.org/wiki/Zipf's_law
- Stephen Robertson and Hugo Zaragoza, **The Probabilistic Relevance Framework: BM25 and Beyond**, Foundations and Trends in Information Retrieval, 2009. http://dx.doi.org/10.1561/1500000019

Implementations

- Apache Lucene, https://lucene.apache.org
- Apache OpenNLP, https://opennlp.apache.org/
- Natural Language Toolkit, https://www.nltk.org/
- Apache Solr, https://solr.apache.org/
- Elasticsearch, https://www.elastic.co/
- OpenSearch, https://opensearch.org/