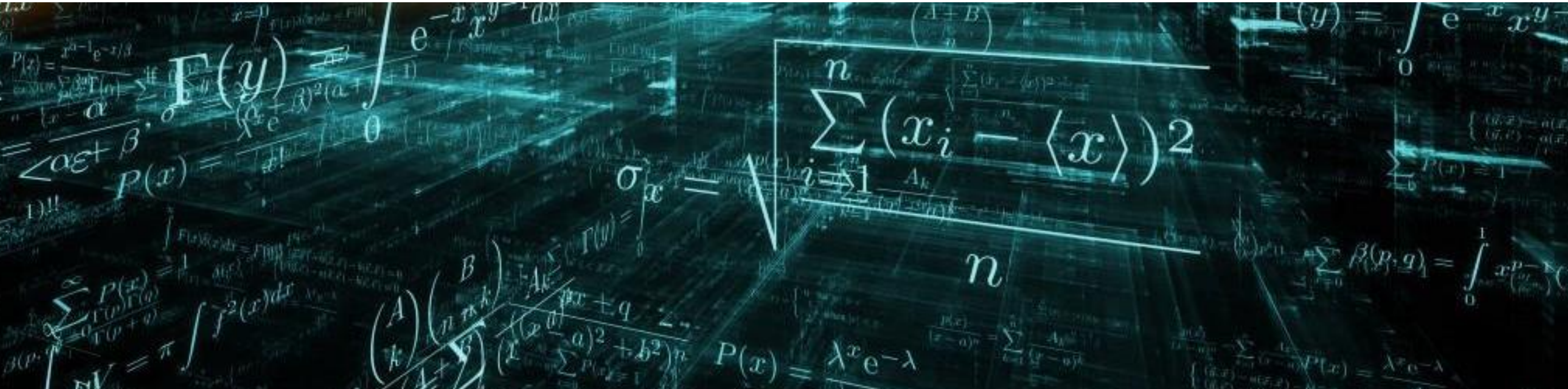
Universität Basel

Multimedia Retrieval

Chapter 5: Indexing Structures for Text Retrieval

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

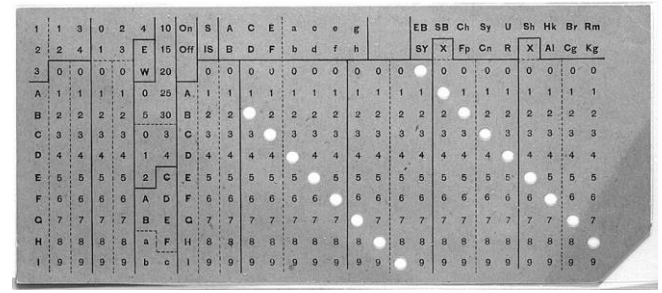


5.1 Introduction

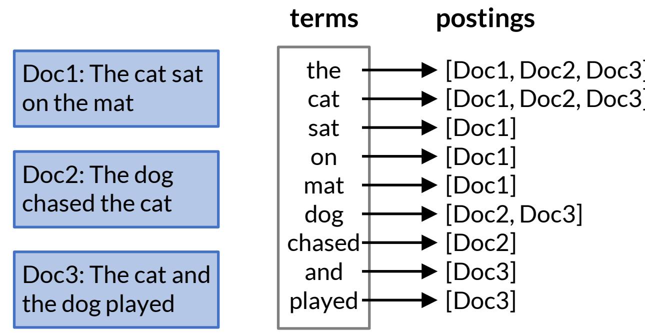
- Text retrieval is a central problem in computer science and supports modern search engines, digital libraries, and information systems. The rapid growth of digital content has changed how we organize, index, and find text, so efficient indexing methods are essential for handling large document collections. Text retrieval is about quickly finding relevant documents in massive datasets from user queries, and it has moved from simple keyword matching to advanced semantic understanding.
- Unlike other media where low-level features do not match high-level concepts, text retrieval benefits from a direct correspondence between user queries and document content because both use natural language. This allowed relatively simple computer systems in early days to provide efficient and effective search for expert users. As computing power improved, retrieval models moved from basic Boolean matching to vector space and probabilistic approaches, and culminated in BM25. The inverted index soon became the dominant data structure for text retrieval because it handles large collections efficiently and supports many retrieval models. Its main advantage is that processing cost grows with the number of query terms rather than with the total size of the document collection.
- Text retrieval has evolved around a basic trade-off among storage efficiency, query speed, and the relevance of results. Early systems focused mainly on exact keyword matching, low costs, and fast results. Modern approaches add linguistic processing, semantic understanding, and machine learning. Frameworks such as Apache Lucene have made industrial-grade text search widely available, letting developers build scalable search applications without writing complex index structures from scratch.
- Text retrieval today covers many uses, from web search engines that handle billions of queries a day to systems for scientific literature, legal documents, and company knowledge bases. Artificial intelligence and natural language processing have expanded what these systems can do, helping them understand user intent, handle synonyms and words with multiple meanings, and return contextually relevant results. As information needs grow more complex, understanding the basic principles of text retrieval indexing is essential for building the next generation of search systems that serve users across fields.
- In this chapter, we study efficient indexing structures for text retrieval and describe algorithms to retrieve results under various classical models. We also examine practical frameworks such as Whoosh, Haystack, and Lucene, and how to scale text retrieval to large-scale document collections with high query volumes. The techniques covered here form the foundation of the AI-enhanced methods used by Google, Perplexity, and ChatGPT.

The Origins of Indexing (1960s-1990s)

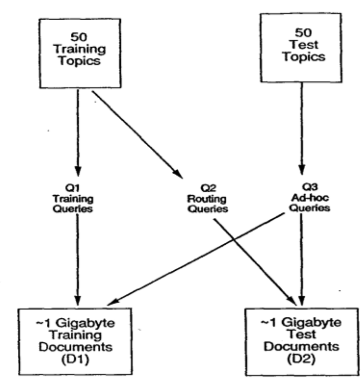
- The history of text retrieval indexing runs from ancient catalogues to today's search engines. It began with Callimachus in the third century BC, who made the first known library catalogue. Mechanization started in the 1880s with Herman Hollerith's punched-card systems, which transformed data processing during the 1890 US Census. Research on automated retrieval grew in the 1940s due to wartime intelligence needs. Vannevar Bush's 1945 Memex idea imagined computerized access to information, and Calvin Mooers coined the term information retrieval in 1950.
- The 1960s were a turning point because of Gerard Salton's work on information retrieval at Harvard. Salton established many of the theoretical foundations for modern text retrieval, including the Vector Space Model and advanced term weighting methods. At the same time, the idea of the inverted index took shape. Instead of listing the terms contained in each document, an inverted index lists, for each term, the documents that contain it. This inversion greatly improved search efficiency and was a major breakthrough for practical use.
- In 1992 the US Department of Defense and the National Institute of Standards and Technology (NIST) co-sponsored the **Text Retrieval Conference (TREC)** under the TIPSTER text program. TREC aimed to provide the infrastructure needed to evaluate text retrieval methods on very large collections and to catalyze research into approaches that scale to huge corpora.
- The TREC initiative created standard evaluation metrics and common benchmarks so researchers could compare different approaches objectively. This led to advanced measures such as Mean Average Precision (MAP), precision at various cutoff levels, and later normalized discounted cumulative gain (NDCG).



Punch card for Herman Hollerith's Electric Sorting and Tabulating Machine, ca. 1895. Source: [Library of Congress]



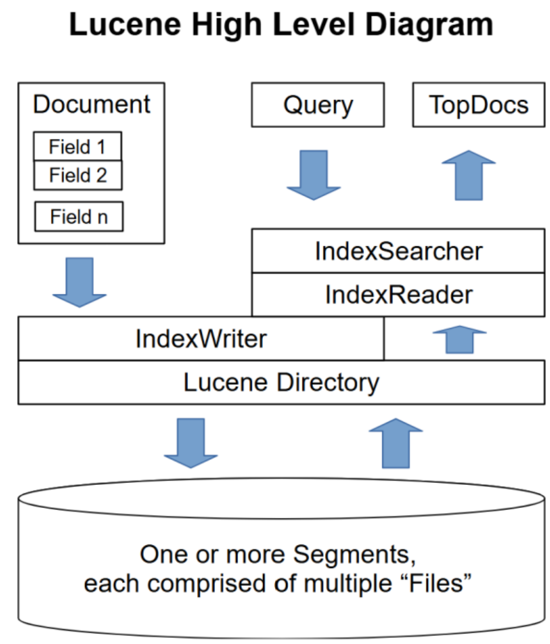
Example of an inverted index



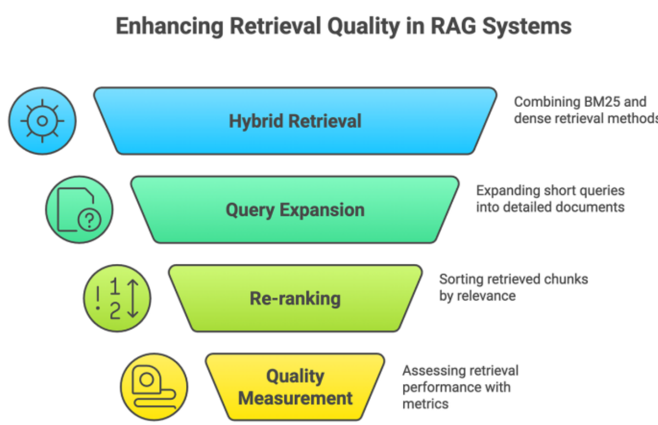
Task description of the first TREC conferences.
Source: [NIST Special Publication]

Modern Indexing Approaches (2000s-)

- When web search engines appeared in the 1990s, they greatly increased the demand for very large retrieval systems. During that time, developers created distributed indexing architectures and improved inverted index structures to handle massive document collections.
- During this period, compression methods for inverted indexes became more important. Using delta compression for document IDs and variable-byte encoding for term frequencies let systems handle larger collections while keeping query times fast.
- Apache Lucene first appeared in 1999, created by Doug Cutting. It made advanced text retrieval available to developers worldwide. Because it is open source and uses inverted indexes along with strong ranking algorithms, it brought industrial-strength search technology within reach. Platforms such as Apache Solr, Elasticsearch, and OpenSearch, all based on Lucene, make horizontal scaling possible across many machines to handle vast amounts of data and concurrent queries.
- The rise of Retrieval-Augmented Generation (RAG) systems in 2024 has renewed interest in classical indexing methods, especially BM25. The development of dense vector representations and neural ranking models has led to hybrid indexing systems that combine the efficiency of inverted indexes with the semantic understanding capabilities of neural networks.
- Today's indexing systems must handle not only massive scale but also real-time updates, multi-modal content, and increasingly sophisticated user expectations for semantic understanding and contextual relevance. The historical progression from manual catalogues to AI-enhanced retrieval systems illustrates the continuous innovation in indexing techniques while maintaining the fundamental principles established in the early decades of information retrieval research.



Source: Lucene.net



Source: medium.com

5.2 Inverted Files

- In all traditional retrieval models discussed so far, we have seen that their scoring functions **depend only on the query terms**. This choice cannot capture deeper semantic relationships between terms, but it is a practical trade off because it allows much faster query processing. Let us explore why this is the case:
  - Assume a collection of N documents and a vocabulary of M terms. On average a document has K distinct terms (much smaller than M), and a typical query contains about $L = 5$ terms (much smaller than K). Each document is represented as a sparse M-dimensional vector, using either the bag-of-words model or the set-of-words model.
  - A simple storage scheme reserves an entry for every term in every document, $N \cdot M$ entries in total. In the set-of-words model, each entry is a single bit indicating whether the term appears. The bag-of-words model needs more space, storing term frequencies using 4, 8, 16, or 32 bits per entry. For example, models such as BM25 use term saturation allowing for term frequency compression to 4 bits without noticeably hurting retrieval accuracy.
  - Scanning all $N \cdot M$ entries for each query is inefficient. Most entries are zero because each document contains only a small fraction, K/M, of all terms. A simple improvement is to store documents in a sparse format, keeping only the K non-zero entries per document. Each entry is either a term ID (for a set-of-words) or a term ID plus its frequency or weight (for a bag-of-words). This reduces storage from $N \cdot M$ to $N \cdot K$ entries, offset by slightly larger entries.
  - Although this sparse representation is more compact, we still must scan all documents at query time. Only a small fraction of that data, specifically the entries for the query terms, affects the final relevance score. Most of what we read never contributes to the result.

- To address this inefficiency, we **flip the storage layout**. Instead of saving for each document the list of terms it contains, we save for each term the list of documents that contain it. This is the main idea of the **inverted index**, also called the **inverted file**. In other words, switch from terms-per-document storage to documents-per-term storage.
  - This change does not alter the total number of stored entries. There are still N times K term-document pairs, but query processing improves dramatically. When a query arrives, we do not need to read all data. We only need to look up the rows, or posting lists, for the query terms.
  - If each term appears, on average, in $N \cdot K/M$ documents, and the query contains L terms, then we only need to read about $N \cdot K \cdot L/M$ entries. Compared to scanning all $N \cdot K$ entries, this yields a reduction by a factor of L/M. For example, if a query has $L = 5$ terms and the vocabulary size $M = 1{,}000{,}000$, we read only **about five-millionths of the data, a huge efficiency gain**.

- An inverted index organizes data by terms rather than by documents. This lets retrieval systems examine only the small part of the data relevant to each query, providing compact storage and fast searches.
  - 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 sub-traction 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 to 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**", retrieve the first entry for each term: document 1 for cat and document 3 for dog. If they match, that document satisfies the query. If not, read the next entry for the term with the smaller document ID. In this example, advance the cat entry to 4. It does not match, so advance the dog entry, which now has the smaller ID. The next dog entry is 4, which matches the cat entry. The first matching document is 4, so we return it as a result for our query.

    For the next result, we fetch the next postings for both terms and repeat the process. Eventually we identify 10 as the second answer. Then we fetch the following posting for both terms. Because the cat's postings are exhausted, we stop the evaluation and end the iteration. Even though dog still has postings, the lack of cat postings means no remaining document can match. The table below shows this approach step-by-step:

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):

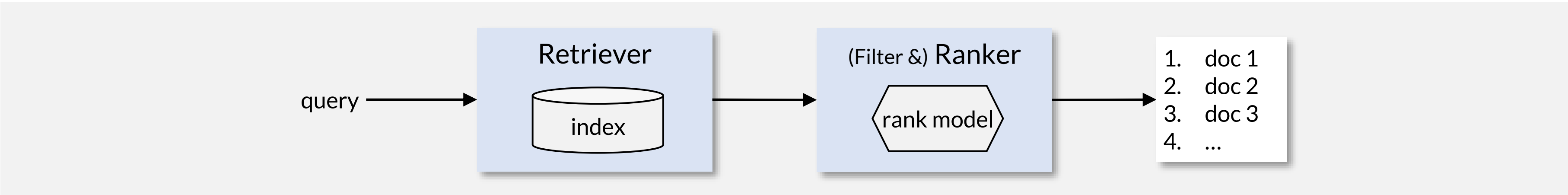
step	cat (next)	dog (next)	action
1	1	3	match, return 1 as result, and progress cat
2	4	3	match but cat is not smallest, so we progress dog
3	4	4	no match as both have the same value, so we progress both cat and dog
4	8	10	match, return 8 as result, and progress cat
5	10	10	no match as both have the 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 for each document the overall score using one of the score combining functions.

5.2.1 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 if k is chosen sufficiently large (e.g., k=1000)
  - The **term-at-a-time** approach processes query terms one by one. For each term it updates the candidate document list and adjusts document scores using the scoring function based on that term's presence. After all terms are processed, documents are sorted by their final scores to produce the ranked list. Unlike document-at-a-time, this method cannot maintain a top-k list during processing to reduce memory use. It can also produce very long candidate lists when the query includes common terms with long posting lists. One optimization is to skip frequent terms that are unlikely to change the ranking significantly.

- 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 repeated sorting after each iteration)
  - 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
    # returns a list of pairs (term_j, c_j)
    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 to reduce computational complexity for sorted access of the result lit.

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

    # filter terms and obtain c_j-weights
    # returns a list of pairs (term_j, c_j)
    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 with 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 (by excluding candidates that are provably not in top-k).
- **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. **A priori filtering:** Store the document attributes (metadata) used in predicates in an external index with an efficient evaluation plan, such as a database. For example, the predicate "year < 2000" can be resolved by an index lookup that returns the document IDs that satisfy it. The index can be a B tree or an inverted list.
     - The best approach for combining text retrieval and predicate filtering is to first retrieve all document IDs that meet the predicate, then pass that set as an inverted list into the search function.
     - Inside the search function, filter out any candidates not in that predicate set. In code, this corresponds to the place where non relevant documents are removed during feedback.
     - Other than evaluating the predicate, the search algorithm does not become more complex.
  2. **A posteriori filtering**: If there is no index for the condition, or if evaluation requires scanning all document data, we check the predicate for each candidate when we return results (in Python `yield`) from the `TopKList` object.
     - `TopKList`'s heap produces a stream sorted by decreasing score. Before handing an object to the caller, we evaluate the document's predicate by accessing its metadata at random. If the predicate fails, we skip that document and take the next one from the heap.
     - In the best case, with a less selective predicate, we only evaluate the predicate for the documents we return and for a few that are skipped. In the worst case, with a highly selective predicate, we must evaluate the predicate for a large portion of the documents in the heap. This is still better than evaluating the predicate across all documents.

3. **Inline filtering:** store metadata in a separate file aligned with the posting lists and ordered by document ID. Add this metadata file to the stream list in the code and advance it in the same way as the other streams
   – In the code, extend the if clause for relevance feedback checks to include a check for the predicate.
   – Continue to use the smallest document ID from the postings for the next iteration, and advance the metadata stream to that same ID.
   – Reading all metadata for every query with predicates can be costly. One optimization is to maintain a stream per metadata attribute and evaluate predicates to produce a stream of only the documents that satisfy the predicate, which is similar to a priori filtering.

5.2.2 Inverted Files for the Vector Space model (and BM25)

- 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.

Alternatively, we can store the extra document information with each posting. This may increase storage and scanning costs but gives more flexibility to adjust scoring parameters.

5.2.3 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 whose leaf nodes hold document IDs and term frequencies, implementing an inverted index inside the database.
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.

```
-- 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.

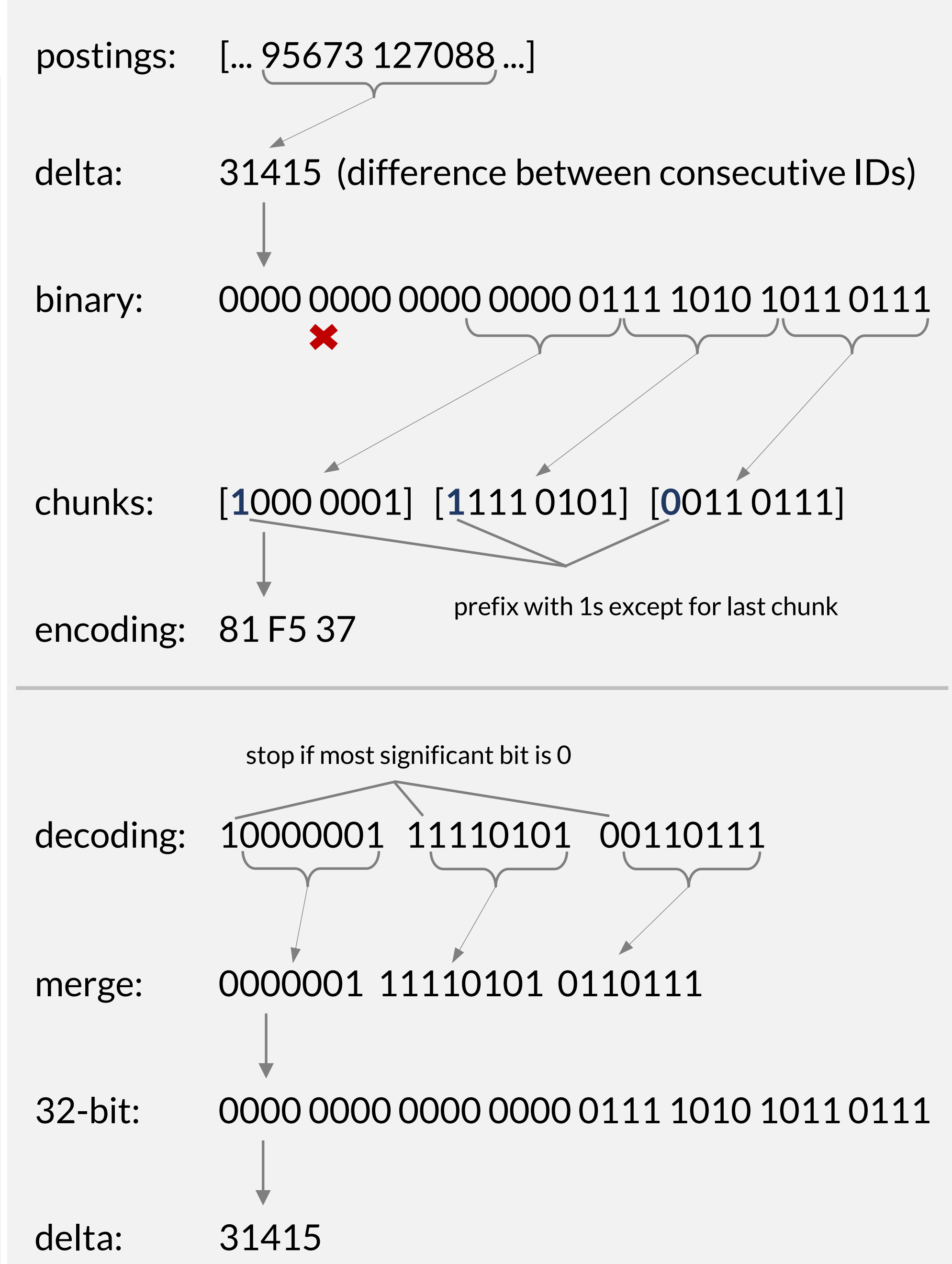
8. Consider how we added predicates to the Python retrieval code using a priori, a posteriori, and inline filtering. Each method has advantages and limitations. In a database, we can evaluate predicates and retrieve text in the same query and let the database engine choose the best execution plan. The example on the right shows vector space retrieval with the predicate "year > 1990".

```
-- 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
```

```
-- 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
```

5.2.4 Compression and Metadata

- Index compression is a key optimization for modern information retrieval systems. Effective compression can cut storage by a factor of four or more and improve query performance because more index data fits into memory and reading data from disk is faster.
- The main idea behind index compression comes from posting lists in inverted indexes. Posting lists store document identifiers in sorted order, so the **differences between consecutive identifiers**, called delta gaps or d-gaps, are usually small integers.
  - **Variable-Byte (VByte)** encoding is the most widely used byte-aligned compression method for posting lists. It uses the most significant bit of each byte as a continuation flag, while the other seven bits store data. If a number needs multiple bytes, every byte except the last has its MSB set to 1. The final byte has MSB set to 0 to mark the end of the encoded integer.
  - Consider the example on the right with document id 31415. Its 32-bit binary representation is shown in the second row. We split that binary number into 7-bit groups and discard any leading zero groups. For each remaining group we add a 1 as the high bit, except for the final group. The resulting hexadecimal encoding is: 81 F5 37.
  - To decode, read bytes while their high bit is set. Then combine the bytes' lower seven bits into a 32-bit value, padding the high bits with zeros. Interpreting that value in decimal gives 31415.



- Other compression approaches:
  - **PForDelta (Patched Frame-of-Reference Delta)** is an advanced compression method that handles integers in fixed-size blocks, usually 128 consecutive posting deltas. For each block it finds the smallest number of bits b that can represent about 90% of the values; those are called regular values. The remaining roughly 10% are stored as exceptions using full 32-bit values. The key idea is to use the unused b-bit slots to build linked lists that point to the exception values. This yields better compression than Variable-Byte encoding.
  - **Byte-aligned compression** puts length markers in the first two bits of each encoded value to show how many bytes the value uses. A common format uses these two-bit markers: 00 for one byte (values 0 to 63), 01 for two bytes (64 to 16,383), 10 for three bytes, and 11 for four bytes. This keeps data aligned to bytes for faster processing and produces compressed indexes about 15 to 20 percent the size of the uncompressed index.
- Adding Metadata
  - Modern inverted indexes store **extra metadata with document identifiers** to support advanced queries and ranking. Term frequency metadata records how often a term appears in a document, enabling TF-IDF scoring and other relevance measures. This data is usually stored as small integers after each document ID in the posting list and compressed with the same methods used for document identifiers.
  - **Document length normalization** data stores precomputed values used by ranking functions such as BM25. This includes document lengths and field-specific statistics. Instead of computing these values during query processing, they are precomputed and kept in compressed structures. This uses more storage but reduces work at query time.
- Support for Faceted Search
  - **Faceted search** needs categorical metadata so users can filter results by attributes like date ranges, geographic locations, or content types. An inverted index for faceted search stores a separate posting list for each facet value, so the system can quickly intersect those lists when processing queries.
  - **Clustered indexing** group similar documents by shared facet values, allowing more efficient compression by using redundancy between related posting lists. If documents in a cluster share many terms, their posting lists can be encoded relative to a reference list, yielding significant space savings.
  - **Hierarchical facet** compression uses the relationships among facet values to reduce storage. For example, geographic facets form a hierarchy like Country > State > City, so compressed representations store only the differences between levels instead of the full path for each item.

5.3 Practical Frameworks

- Modern information retrieval applications require reliable frameworks that hide the complexity of indexing and searching while providing the scalability and performance needed in production. In the rest of this chapter we review several practical text retrieval frameworks, each showing a different stage in the evolution of an application, from lightweight experimental tools to distributed, enterprise-grade systems. In this section:
  - We start with Whoosh, a pure Python search engine built for simplicity and ease of use. Whoosh is self-contained and easy to install, and it does not require external services or Java. Its index format and analyzer pipeline are written entirely in Python, making it suitable for learning, testing ranking algorithms, and adding search to small applications. While Whoosh does not provide the distributed scale of larger systems, it stands out for its clarity and transparency, which makes it a popular choice for teaching and rapid prototyping.
  - Next, we turn to **PostgreSQL Full Text Search (FTS)**, a feature-rich part of the database that adds full-text search directly to SQL. PostgreSQL uses the `tsvector` and `tsquery` types to tokenize, normalize, and match text inside relational data. It supports stemming, stop words, ranking functions such as `ts_rank`, and GIN and GiST indexes for fast lookups. This integration lets developers build consistent, transactional systems with search living alongside the rest of the data model. It is an elegant choice to avoid running a separate search system.
  - Next, we consider **Haystack**, a higher-level framework that provides an abstraction layer for different search backends. Haystack offers a single API for indexing and querying data, no matter the underlying engine or backend service (including cloud-based services). This abstraction lets you start with a lightweight backend like Whoosh for development and then move to a more powerful one in production.

  Together these frameworks reflect the goals of early-stage search development. They are lightweight, adaptable, and approachable, suited to small and mid-size applications, research prototypes, and teaching.

- The next section looks at **Lucene**, a Java-based core library that enables most of today's advanced search platforms. Lucene supplies the basic tools of modern information retrieval: tokenization, inverted indexing, term weighting, and ranking methods. It is highly configurable and lets developers create custom analyzers, filters, and scoring models.

- In the final section, we examine **Solr**, **Elasticsearch**, and **OpenSearch**. These distributed systems build on Lucene and add clustering, horizontal scalability, load balancing, and real time search. Solr uses a schema-driven enterprise model and supports faceting, multilingual text, and integration with existing data platforms. Elasticsearch and OpenSearch use a schema-flexible RESTful API that emphasizes easy integration, analytics, and near real time updates.

Framework	Type	Core Engine	Language	Deployment Complexity	Scalability	Typical Use Case
Whoosh	Lightweight full-text search library	Pure Python	Python	Very low	Low	Embedded search in small apps
PostgreSQL Full Text Search (FTS)	Database-integrated search	PostgreSQL (SQL-based)	SQL / any client	Moderate	Moderate	Searching text stored in relational DB
Haystack	Search abstraction layer	Pluggable (Whoosh, Elasticsearch, etc.)	Python	Depends on backend	Depends on backend	Unified search interface for Django apps
Lucene	Core text search library	Java	Java	High (requires embedding)	High	Building custom search systems
Elasticsearch OpenSearch	Distributed search engine	Lucene	REST / JSON	Medium-High	Very High	Large-scale distributed search
Solr	Enterprise search platform	Lucene	Java / HTTP API	Medium-High	Very High	Enterprise / content management search

- **Whoosh** is a pure Python library for full-text search that makes it easy to learn information retrieval and quickly prototype search applications:
  1. Whoosh requires you to **specify schema fields** before you start indexing, like Lucene but using Python syntax. Fields can be set to TEXT for tokenized full-text search, KEYWORD for exact matches, ID for unique identifiers, or STORED to keep values without indexing.
  2. Whoosh's **inverted index** follows standard information retrieval methods and exposes its internals to Python. For each term, the library builds a posting list that records which documents contain the term, along with term frequency and position data, based on the field settings. Whoosh's real-time updates add, delete, and modify documents through a writer interface that manages index segments and applies changes incrementally.
  3. Whoosh supports **several scoring methods**, including term frequency counting, TF-IDF, BM25F, and cosine similarity. BM25F builds on basic BM25 to work with multiple document fields and different weights, so a match in a title can score higher than the same match in an overview field. Whoosh lets you build queries in code or parse them from strings. The parser handles Boolean operators AND, OR, NOT; quoted phrase searches; wildcards with * and ?; and field-specific searches using `field:term`. Fuzzy search can match misspelled terms within a specified edit distance.

```python
from whoosh import index, scoring
from whoosh.fields import Schema, TEXT, ID, NUMERIC
from whoosh.qparser import MultifieldParser
from whoosh.query import And, NumericRange

# -- 1. Define a schema for movies ---
schema = Schema(
    id       = ID(stored=True, unique=True),
    title    = TEXT(stored=True),
    overview = TEXT(stored=True),
    rating   = NUMERIC(stored=True, numtype=float),
    year     = NUMERIC(stored=True, numtype=int)
)


# -- 2. Index movie objects ---
ix = index.create_in("movie_index", schema)
writer = ix.writer()
for m in movies:
    writer.add_document(**m)
writer.commit()


# -- 3. Search using BM25 and predicate ---
weighting = scoring.BM25F(title_B=2.0)
with ix.searcher(weighting=weighting) as searcher:
    # Search across both title and overview
    parser = MultifieldParser(["title", "overview"],
                              schema=ix.schema)
    query = parser.parse("Star Wars")

    # Filter results by year < 2000
    combined_query = And([query,
                          NumericRange("year", None, 2000)])

    # Run the search
    results = searcher.search(combined_query, limit=10)
    for r in results:
        print(f"{r['title']} ({r['year']})")
```

- **PostgreSQL full-text search (FTS)** capabilities demonstrate how relational databases have evolved to incorporate information retrieval functionality:
  1. PostgreSQL uses the `tsvector` and `tsquery` data types for full-text search. `tsvector` stores text as a normalized list of lexemes. `tsquery` stores search expressions that allow Boolean operators and phrase matching. The function `to_tsvector` converts plain text into a `tsvector` by applying stemming, removing stop words, and optionally assigning weights to parts of the text to affect ranking. The function `to_tsquery` converts a search string into a `tsquery`.
  2. **GIN (Generalized Inverted Index)** is the main index type used for PostgreSQL's full-text search. `GIN` builds an inverted index that maps each lexeme to the documents containing it, speeding up full-text queries. The index uses a B-tree at the top level and compressed posting lists for each lexeme, saving space while keeping query performance high. This process closely follows the inverted index construction algorithms described earlier.
  3. The match operator (`@@`) tests whether a document matches a query. PostgreSQL uses the `ts_rank` function for ranking and relevance scoring; it implements BM25-style scoring with document length normalization. A key advantage is its integration with relational queries, allowing complex queries that combine full-text search with standard SQL.

```sql
-- 1. Define a table for movies
CREATE TABLE movies (
    id          SERIAL PRIMARY KEY,
    title       TEXT NOT NULL,
    overview    TEXT NOT NULL,
    rating      REAL,
    year        INT,
    tsv         tsvector
);



-- 2. Populate the tsvector column for full-text search
--    'A' = highest weight (title gets more influence)
--    'B' = lower weight (overview contributes less)
UPDATE movies
   SET tsv = setweight(to_tsvector(title), 'A') ||
             setweight(to_tsvector(overview), 'B');

-- (optional) create an index for performance
CREATE INDEX movies_tsv_idx
          ON movies
       USING GIN(tsv);



-- 3. Search using BM25-style ranking and predicate on year
SELECT
    id,
    title,
    year,
    ts_rank(tsv, to_tsquery('Star & Wars')) AS r
FROM movies
WHERE
    tsv @@ to_tsquery('Star & Wars')
    AND year < 2000
ORDER BY r DESC
LIMIT 10;
```

- **Haystack** is a modern framework for building retrieval-augmented generation (RAG) pipelines and search systems. Its modular, pipeline-based design connects reusable components into flexible processing chains that handle both traditional keyword search and modern neural retrieval methods.
  1. **Document Stores** serve as the foundation of Haystack's architecture, providing persistent storage for indexed documents and their associated metadata. The framework supports multiple document store implementations, from simple in-memory stores (`InMemoryDocumentStore`) for development to production-grade solutions like Elasticsearch and vector databases.
  2. **Retrievers** run search algorithms that find documents relevant to user queries. Haystack provides several retriever types. `InMemoryBM25Retriever` uses keyword search with the BM25 ranking function. It follows the document at a time evaluation strategy described earlier and searches the full list of documents.

  Haystack is commonly used to build longer pipelines composed of stores, embedders, retrievers, rerankers, generators, and other components to integrate different implementations. Pipeline orchestration links these parts into end-to-end workflows using explicit connection definitions. Those definitions make it easier to debug, test, and modify search workflows.

```python
from haystack import Document
from haystack.components.retrievers.in_memory
        import InMemoryBM25Retriever
from haystack.document_stores.in_memory
        import InMemoryDocumentStore


# --- 1. Create a document store ---
document_store = InMemoryDocumentStore()
movies = [
    {"id": "1", "title": "Star Wars: A New Hope", ...}
    # more data
]
movies_documents = [
    Document(
        content=f"{m['title']} {m['overview']}",
        meta={ "id": m['id'], "title": m['title'],
               "year": m['year'], "rating": m['rating']}
    )
    for m in movies
]
document_store.write_documents(movies_documents)


# --- 2. Perform a search with a filter on year < 2000 ---
retriever = InMemoryBM25Retriever(
                  document_store=document_store, top_k=10)

results = retriever.run(
    query="Star Wars",
    filters={"field": "meta.year",
             "operator": "<",
             "value": 2000}
)

for r in results:
    print(f"{r.meta['title']} ({r.meta['year']})")
```

5.4 Lucene - Opens Source Text Search

- **Lucene**, initiated by Doug Cutting in 1999, 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), 2024 (v10)
  - Major releases may overlap in time, 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:

- 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.

```java
// 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.
   - `fuzy~0.6` conducts a search for terms closely related (0.6 defines the closeness) to "fuzy". 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("fuzy~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:

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

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

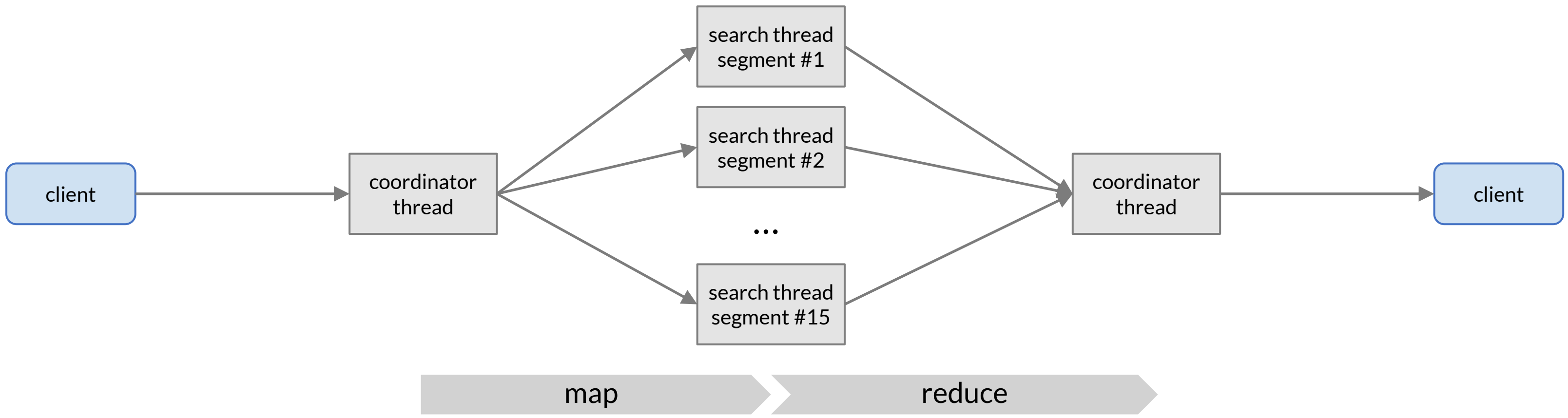
  - `n` represents the document frequency, calculated individually for each field.
  - `boost` is an additional factor for the query part (as seen in `qBody` with `boost = 1.5`).
  - `k1` and `b` are parameters for term frequency normalization in line with the BM25 formula. Document length (`dl`) and average 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
```

5.5 Scaling Text Retrieval

- 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.

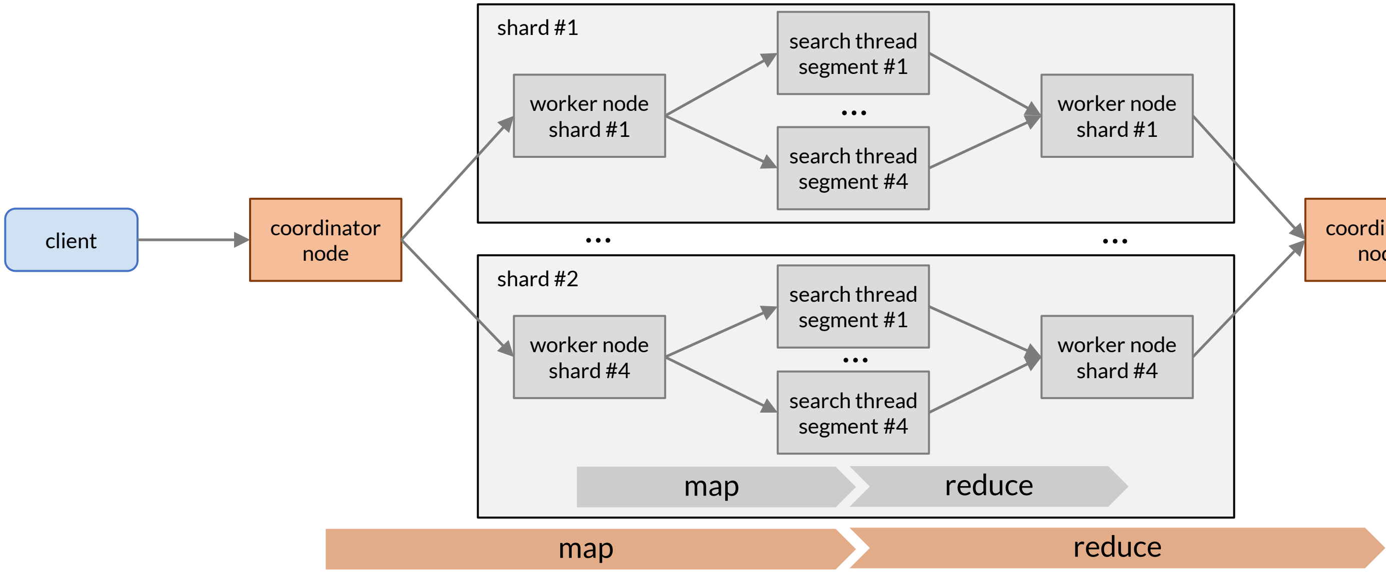


     Applying Amdahl's law and assuming 99% of the time is spent on segment searching, we can achieve a 13-fold speedup compared to a single thread. Nevertheless, search times increase proportionally with the index size (until Lucene's limitations are reached), and we can only execute one query at a time 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.

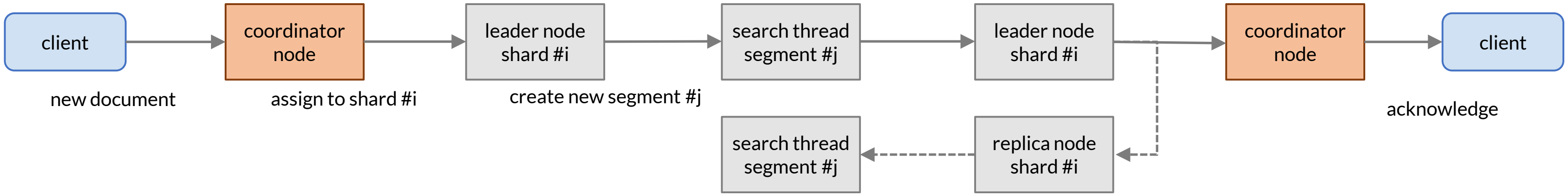
   When a new document is introduced to the collection, a coordinator node determines the shard to which it is assigned. This assignment can be based on predefined policies like round-robin, or by utilizing user-defined prefixes on the document ID which are hashed to a specific shard number (ensuring that 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.

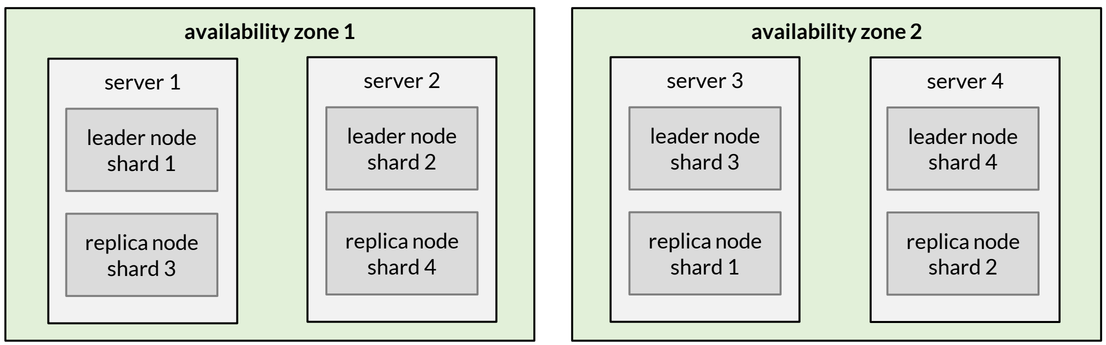


   With this configuration, we can lower our hardware demands and utilize numerous smaller worker nodes. These nodes can be operated within a Kubernetes cluster distributed across physical servers. Rapidly deploying new worker nodes and responding promptly to node failures enhances overall availability,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

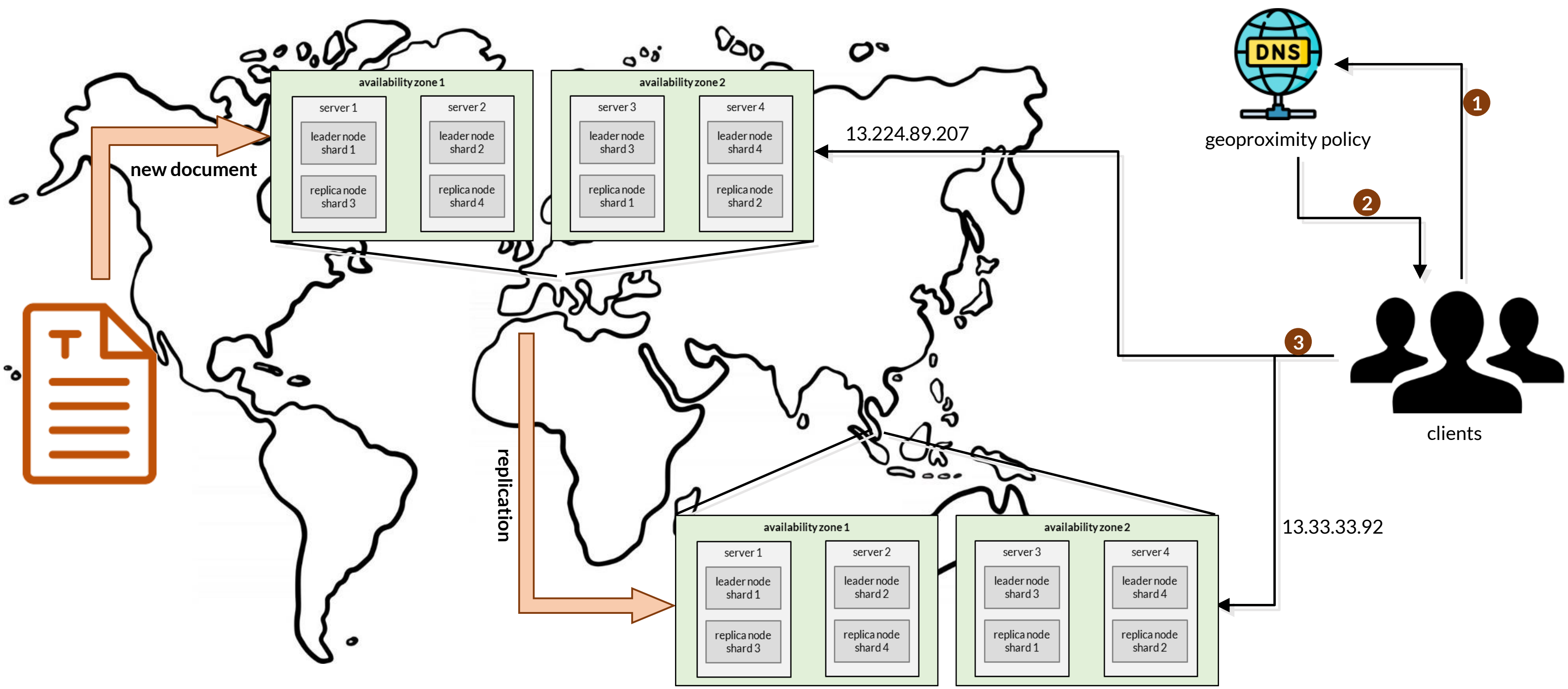


The number of replicas influences overall system availability and the system's resilience against failures. In our example with 4 shards and 2 replicas distributed across 2 availability zones, it might look as follows (each leader node can take over the role of the coordinator role as well):



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)

5.6 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.

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/
- PostgreSQL Full Text Search (FTS), https://www.postgresql.org/docs/current/textsearch.html
- Whoosh, https://whoosh.readthedocs.io/en/latest
- Haystack, https://haystack.deepset.ai