Information Retrieval and Text Mining

Indexing techniques

Nuno Escudeiro (nfe@isep.ipp.pt)
Ricardo Almeida (ral@isep.ipp.pt)

Session outline

1. Indexing in IR
2. Performance issues
3. Indexing techniques
4. Compression
5. Parallel and distributed processing

Learning outcomes

At the end of this session, we will be able to:

- Describe the purpose, objectives and fundamentals of the indexing process in IR
- Enumerate end explain performance issues of indexing in IR
- Identify and describe tools for index compression and distributed indexing

1. Indexing in IR

A general overview of the Indexing process in IR

Indexing in IR: what?

- Process of **organizing and storing** documents in a way that enables efficient search and retrieval of information based on keywords
- Involves **creating data structures** that map terms (words or phrases) to the documents in which they appear
- It has a fundamental role in IR systems related to **speed and efficiency of searches**

Can you imagine a search engine that needs 1 minute to show you the results of a query!?.

Indexing in IR: why?

Objectives

- **Fast Retrieval:** indexing allows users to quickly locate relevant documents based on their search queries

Indexing in IR: why?

Objectives

- **Fast Retrieval:** indexing allows users to **quickly** locate relevant documents based on their search queries

"**between 200 milliseconds and 1 second** is considered acceptable as users still likely won't notice the delay".

https://sematext.com/glossary/response-time/

Indexing in IR: why?

Objectives

- **Fast Retrieval:** indexing allows users to quickly locate relevant documents based on their search queries
- **Reduced Search Space:** instead of scanning through the entire document collection for each search query, indexing helps narrow down the search to documents containing specific terms
- **Support for Complex Queries:** indexing enables the execution of complex queries involving multiple terms, Boolean operators, and proximity constraints
- **Scalability:** indexing techniques are designed to handle large volumes of documents efficiently.

2. Performance issues

When indexing a corpus, several issues need careful consideration to ensure optimal performance

Critical issues to consider

Indexing a corpus impacts the performance of IR:

- **efficiency,**
- **accuracy and**
- **scalability of search operations**

To control such impacts, several issues need careful consideration to ensure optimal/good performance when indexing a corpus.

Critical issues to consider

Indexing a corpus impacts the performance of IR:

- **efficiency,**
- **accuracy and**
- **scalability of search operations**

To control such impacts, several issues need careful consideration to ensure **optimal/good performance** when indexing a corpus.

Critical issues to consider

Good performance:
- Fast Retrieval
- Reduced Search Space
- Support for Complex Queries
- Scalability

Indexing a corpus impacts the performance of IR:

- **efficiency,**
- **accuracy and**
- **scalability of search operations**

To control such impacts, several issues need careful consideration to ensure **optimal/good performance** when indexing a corpus.

Critical issues to consider

1. Vocabulary Management
2. Scalability and Performance
3. Indexing Granularity
4. Document Representation
5. Handling Multi-Language and Multilingual Corpora
6. Efficient Query Processing
7. Handling Noisy Data
8. Security and Access Control
9. Compression and Storage
10. Fault Tolerance and Consistency
11. Maintenance and Updates
12. Evaluation and Quality Metrics

Critical issues to consider

1. Vocabulary Management

- **Synonyms and Polysemy**: Handle terms with multiple meanings (polysemy) and different terms with the same meaning (synonyms)
    - Solution: stemming, lemmatization, controlled vocabularies, lists of synonyms, thesaurus, lexical databases (such as Wordnet)
- **Stop Words**: Decide whether to remove common stop words (like "the," "is") to save space, but balance it against potential loss of context
- **Case Sensitivity**: Determine whether to make indexing case-insensitive for uniformity.

Critical issues to consider

2. Scalability and Performance

- **Index Size**: As the corpus grows, the index size can become very large
  - Solution: compression techniques like delta encoding or variable-byte encoding
- **Real-Time Indexing**: If dealing with frequently updated data, ensure real-time or near-real-time indexing without compromising performance
- **Concurrency**: Design for multi-threading or parallel processing, especially in distributed systems.

Critical issues to consider

3. Indexing Granularity

- **Term versus Phrase Indexing**: Decide whether to index individual words (terms), N-Grams or phrases for more precise querying
- **Positional Indexing**: Store positional information for proximity searches or phrase queries, increasing index size but improving retrieval quality.

Critical issues to consider

4. Document Representation

- **Structured vs. Unstructured Data**: Address indexing differences between structured fields (like metadata) and unstructured content (like text).
- **Metadata Indexing**: Index additional fields like author, title, date, etc., to enable advanced search capabilities.
- **Structural features**: Decide whether to model document structure (hyperlinks, images, chapters/sections, paragraphs)
- **Numerals and Punctuation**: Decide whether to remove numerals and punctuation.

Critical issues to consider

5. Handling Multi-Language and Multilingual Corpora

- **Character Encoding**: Ensure consistent encoding (like UTF-8) for diverse languages/alphabets; decide whether to consider orthography aspects like national diacritical marks (accents: “c”, “ç”)
- **Language Detection**: Use language detection tools to apply language-specific preprocessing (stemming, stop word removal)
- **Cross-Language Retrieval**: If necessary, incorporate techniques for cross-lingual indexing and retrieval.

Critical issues to consider

6. Efficient Query Processing

- **Efficient Data Structures**: Choose efficient structures like inverted indexes, B-trees, or skip lists for faster access.
- **Caching Mechanisms**: Implement query caching for frequent searches to reduce latency.
- **Ranking and Scoring**: Define scoring mechanisms (like TF-IDF, BM25) for relevance-based ranking.

Critical issues to consider

7. Handling Noisy Data

- **Typos and Misspellings**: Use approximate matching techniques like n-grams or edit distance (Levenshtein distance)
- **Data Cleaning**: Remove irrelevant or corrupt data to improve the signal-to-noise ratio (numerals?, punctuation?, diacritical marks?)
- **Normalization**: Normalize data for consistent indexing (remove punctuation, standardize spellings).

Critical issues to consider

8. Security and Access Control

- **Access Control**: Implement role-based access or authentication for sensitive data
- **Data Privacy**: Ensure compliance with data protection regulations (e.g., GDPR) to avoid indexing sensitive personal information.

Critical issues to consider

9. Compression and Storage

- **Compression**: Use index compression techniques (like Variable-Length Encoding or Delta Encoding) to balance size and speed
- **Disk versus Memory**: Optimize index placement based on available memory; keep frequently accessed data in memory.

Critical issues to consider

10. Fault Tolerance and Consistency

- **Backup and Recovery**: Implement backup strategies for disaster recovery.
- **Index Consistency**: Ensure index consistency, especially in distributed systems, to avoid stale or corrupted data.

Critical issues to consider

11. Maintenance and Updates

- **Dynamic Indexing**: Support efficient updates, deletions, and incremental indexing.
- **Re-indexing**: Plan for periodic re-indexing if data changes significantly to maintain relevance (web crawling cycles).

Critical issues to consider

12. Evaluation and Quality Metrics

Relevance and Precision: Regularly evaluate retrieval effectiveness using metrics like precision, recall, and F1-score

Latency and Throughput: Monitor system performance to ensure low-latency retrieval

Critical issues to consider

12. Evaluation and Quality Metrics

Relevance and Precision: Regularly evaluate retriev
metrics like precision, recall, and F1-score

Latency and Throughput: Monitor system performan
latency retrieval

How to measure Relevance and Latency?

Relevance

- Record user interaction (hits on search results)
- Implement some kind of explicit relevance feedback by users

Latency

- Record latency per query.

3. Indexing Techniques

Indexing techniques in IR help efficiently retrieve relevant documents from a large collection based on user queries.

Indexing techniques

1. Inverted Indexing
2. Forward Indexing
3. Signature Files
4. Suffix Trees
5. n-Gram Indexing
6. Bit-Sliced Indexing
7. Latent Semantic Indexing (LSI)
8. Bitmap Indexing
9. B-Tree and B+-Tree Indexing
10. Skip Lists
11. Distributed and Parallel Indexing
12. Hierarchical and Cluster-Based Indexing
13. Vector Space Indexing
14. Neural Embedding-Based Indexing

Indexing techniques

1. Inverted Indexing

- The most popular indexing technique in IR
- Maps terms (words) to their occurrences in documents, allowing quick lookup

2. Forward Indexing

- Maps documents to the list of terms they contain
- Less efficient for querying but useful for initial data organization.

Indexing techniques

3. Signature Files

- Uses hashing to create a compact representation (signature) of documents.
- Efficient for membership tests but may lead to false positives

4. Suffix Trees

- Indexes all possible suffixes of a text for efficient substring searches
- Commonly used in bioinformatics and text retrieval with wildcards.

Indexing techniques

5. n-Gram Indexing

- Breaks text into contiguous sequences of 'n' characters or words
- Useful for approximate matching and handling spelling errors

6. Bit-Sliced Indexing

- Represents terms as bit slices across documents
- Efficient for Boolean and range queries
- Especially useful for large datasets with repeated values.

Indexing techniques

7. Latent Semantic Indexing (LSI)

- Uses Singular Value Decomposition (SVD) to reduce dimensions of term-document matrices
- Captures underlying semantic structures for improved relevance

8. Bitmap Indexing

- Uses bitmap vectors to represent the presence or absence of terms in documents → Boolean model
- Effective for categorical and low-cardinality (categorical) data.

Indexing techniques

9. B-Tree and B+-Tree Indexing

- Hierarchical tree structures for ordered data
- Effective for range queries and databases with numeric data

10. Skip Lists

- Ordered linked lists with pointers skipping several elements organized in distinct levels that are searched from sparse to dense levels
- Efficient for range-based searches and sorted data.

Indexing techniques

11. Distributed and Parallel Indexing

- Utilized in large-scale distributed IR systems like search engines
- Techniques like MapReduce are applied for distributed indexing

12. Hierarchical and Cluster-Based Indexing

- Organizes documents in clusters or hierarchical structures
- Speeds up searches by navigating through grouped data.

Indexing techniques

13. Vector Space Indexing

- Represents documents and queries as vectors
- Uses cosine similarity for matching and ranking results

14. Neural Embedding-Based Indexing

- Utilizes word embeddings like Word2Vec or BERT for semantic indexing
- Effective for contextual and semantic search tasks.

Indexing techniques

1. Inverted Indexing
2. Forward Indexing
3. Signature Files
4. Suffix Trees
5. n-Gram Indexing
6. Bit-Sliced Indexing
7. Latent Semantic Indexing (LSI)
8. Bitmap Indexing
9. B-Tree and B+-Tree Indexing
10. Skip Lists
11. Distributed and Parallel Indexing
12. Hierarchical and Cluster-Based Indexing
13. Vector Space Indexing
14. Neural Embedding-Based Indexing

Indexing techniques

1. Inverted Indexing
2. **Forward Indexing**
3. Signature Files
4. **Suffix Trees**
5. **n-Gram Indexing**
6. Bit-Sliced Indexing
7. **Latent Semantic Indexing (LSI)**
8. Bitmap Indexing
9. B-Tree and B+-Tree Indexing
10. Skip Lists
11. **Distributed and Parallel Indexing**
12. **Hierarchical and Cluster-Based Indexing**
13. **Vector Space Indexing**
14. Neural Embedding-Based Indexing

Forward Indexing

Concept: Opposite of inverted indexing. It maps each document to a list of terms it contains.

Structure: A mapping from document IDs to lists of terms.

Example:

Document 1 → ["data", "search", "retrieval"]

Document 2 → ["index", "structure"]

Advantages: Useful for quick document analysis and metadata retrieval.

Disadvantages: Inefficient for query-based searching; often used alongside inverted indexes.

Suffix Trees

Concept: A tree structure that indexes all possible suffixes of a string. It is a type of trie with each path representing a suffix.

Use Case: Efficient for substring search, pattern matching, and bioinformatics (DNA sequence search).

Example: For the string "banana", the suffix tree contains:

"banana", "anana", "nana", "ana", "na", "a"

Advantages: Efficient for complex substring queries and pattern matching.

Disadvantages: High memory usage for large texts; complex to construct.

N-Grams

Concept: Breaks text into contiguous sequences of 'n' characters or words.

Use Case: Effective for approximate matching, spell-checking, and language modeling.

Example: For the word "text" with n=2 (bigrams), we have:

"te", "ex", "xt"

Advantages: Robust for partial matches and spelling error tolerance.

Disadvantages: Index size grows rapidly with larger 'n'; may increase false positives.

Latent Semantic Indexing (LSI)

Concept: Uses Singular Value Decomposition (SVD) to reduce dimensions of the term-document matrix, capturing latent semantic relationships.

Use Case: Handles synonymy and polysemy, improving semantic search.

Example: A matrix representation of documents and terms is factorized:

$$A \text{ (terms x docs)} = U \text{ (terms x concepts)} \times \Sigma \text{ (concepts)} \times V^T \text{ (concepts x docs)}$$

Advantages: Captures deeper semantic meaning; effective for retrieval in noisy data.

Disadvantages: Computationally expensive; not effective for very large datasets.

Distributed indexes

Concept: Used in large-scale distributed systems like search engines (e.g., Google, Elasticsearch). Data is partitioned across multiple machines, and MapReduce techniques help in parallel processing.

Use Case: Efficient indexing and retrieval in big data environments.

Example:

Sharding: Partitioning index across nodes.

Replication: Ensures fault tolerance.

Advantages: Scales well with massive datasets; high availability and redundancy.

Disadvantages: Complex infrastructure; requires synchronization and consistency mechanisms.

Hierarchical and Cluster-Based Indexing

Concept: Organizes documents into clusters or hierarchies based on similarity, reducing search space.

Use Case: Useful in categorized search and faceted search.

Example: A hierarchical index might look like:

Technology
- Software
- Hardware

Science
- Physics
- Chemistry

Advantages: Faster searches through category navigation; effective in exploratory searches.
Disadvantages: Complex to maintain as data evolves; relies heavily on initial clustering quality.

Vector Space Indexing

Concept: Represents documents and queries as vectors in a high-dimensional space. Similarity is computed using measures like cosine similarity.

Use Case: Effective in ranked retrieval and relevance scoring.

Example: Document vectors:

D1 = [0.2, 0.3, 0.5]

D2 = [0.1, 0.7, 0.2]

Cosine Similarity = (D1 · D2) / (||D1|| ||D2||)

Advantages: Flexible, supports weighted queries and relevance ranking.

Disadvantages: Computationally intensive for large corpora; sparsity issues with large vocabularies.

Indexing techniques

1. **Inverted Indexing**
2. Forward Indexing
3. Signature Files
4. Suffix Trees
5. n-Gram Indexing
6. Bit-Sliced Indexing
7. Latent Semantic Indexing (LSI)
8. Bitmap Indexing
9. B-Tree and B+-Tree Indexing
10. Skip Lists
11. Distributed and Parallel Indexing
12. Hierarchical and Cluster-Based Indexing
13. Vector Space Indexing
14. Neural Embedding-Based Indexing

Inverted Indexes (Basic concepts)

- Is a word-oriented mechanism for indexing a text collection to speed up the searching task – maps terms to the documents in which they appear
- Its structure has two elements:
  - Vocabulary
  - Occurrences

- Is constructed by tokenizing documents, creating a dictionary of terms, and maintaining posting lists for each term
- The simplest way to represent the documents that contain each word of the vocabulary is the Term x Document matrix

Inverted Indexing

Concept: Maps each term in a document collection to a list of documents where the term appears. It's the backbone of modern search engines.

Structure: Typically consists of two components:

Dictionary: The set of unique terms.

Posting Lists: Lists of documents where each term appears, along with positions (for positional indexes).

Example: Term → Document IDs

"data" → [1, 3, 5]

"index" → [2, 3]

"search" → [1, 4, 5]

Advantages: Efficient for fast keyword searches.

Disadvantages: Space-intensive for large vocabularies; requires compression techniques.

Inverted Indexing

Concept: Maps each term in a document collection to a list of do
It's the backbone of modern search engines.

Structure

term appe

Example:

Suppose we have a small collection of three documents:
Doc 1: "information retrieval is essential"
Doc 2: "retrieval techniques improve search"
Doc 3: "information is key to retrieval"

Create the Inverted Index with Posting Lists?

Review to add Frequencies and Positions?

A **posting list** is a fundamental data structure used in information retrieval, especially in **inverted indexing**. It stores information about which documents a particular term appears in, making it efficient to retrieve relevant documents based on keyword searches.

A posting list typically consists of:

- **Term (Keyword):** The word being indexed.
- **Document IDs (DocIDs):** The unique identifiers of documents where the term appears.
- **Term Frequency (Optional):** The number of times the term appears in each document.
- **Positional Information (Optional):** Positions within the document where the term occurs — useful for phrase searches.

Advantages: Efficient for fast keyword searches.
Disadvantages: Space-intensive for large vocabularies; requires compression techniques.

Inverted Indexes - Creation

1. **Tokenization**: Break documents into individual terms (tokens, words, n-grams)
2. **Dictionary Construction**: Create a dictionary of unique terms from the tokenized terms
3. **Occurrences (Posting List)**: For each term, create a posting list containing document IDs where the term occurs.
4. **Index Construction**: Combine the dictionary and posting lists to create the inverted index

Inverted Indexes

- Consider the following documents:
- D1:”To do is to be. To be is to do.”
- D2:”To be or not to be. I am what I am.”
- D3:”I think therefore I am. Do be do be do.”
- D4:”Do do do, da da da. Let it be, let it be.”

Baeza-Yates’ – figura3.6

Inverted Indexes (Basic concepts)

Vocabulary	Number of documents (n_i)	D1	d2	d3	d4	Ocurrences as inverted list
to	2	4	2	0	0	[1,4], [2,2]
do	3	2	0	3	3	[1,2], [3,3], [4,3]
is	1	2	0	0	0	[1,2]
be	4	2	2	2	2	[1,2], [2,2],[3,2],[4,2]
or	1	0	1	0	0	[2,1]
not	1	0	1	0	0	[2,1]
I	2	0	2	2	0	[2,2],[3,2]
am	2	0	2	1	0	[2,2],[3,1]
what	1	0	1	0	0	[2,1]
think	1	0	0	1	0	[3,1]
therefore	1	0	0	1	0	[3,1]
da	1	0	0	0	3	[4,3]
let	1	0	0	0	2	[4,2]
it	1	0	0	0	2	[4,2]

4. Compression

Compression techniques in indexing aim to reduce the storage size of index structures while maintaining quick access

Compression techniques

Compression and parallel/distributed processing are essential techniques for optimizing indexing in IR:

- manage large-scale data efficiently,
- reduce storage costs,
- speed up query processing,
- handle high-volume,
- concurrent access.

Text compression

- Text compression is a technique used to **reduce the storage space required for textual data while preserving its essential Information**
- In information retrieval systems, text compression plays an important role in improving storage efficiency, enabling faster data transmission, and enhancing overall system performance

Text compression

- **Text data is inherently redundant**, containing patterns and repetitions that can be exploited for compression
- By compressing text data it is possible to reduce storage requirements. This is particularly important for large corpora and web-based information retrieval systems
- Facilitates faster data transmission over networks, leading to improved user experience in online search and retrieval.

Text compression: main issue for IR

- The major obstacle for storing text in compressed form in IR is the **need to access text randomly**
- Access a given word in compressed text, **require the decompression from scratch**.

- The text compression methods we will review, **do not require full decompression** for random access!

Text compression: main characteristics

- Economy of space
- Compression and decompression speed
  - Which one is more critical, compression or decompression speed?
- Possibility of searching the compressed text without decompressing it

Text compression approaches

- Aproaches:
  - **Statistical methods** – estimate the probability of appearing of each text symbol
    - Alphabet
    - Model
    - Coding
    - E.g: of Statistical methods: Huffman and arithmetic coding
  - **Dictionary based**
    - E.g: Ziv-Lempel family
  - **Preprocessing** for compression
    - E.g: Burrows-Wheeler

Text compression approaches

- Aproaches:
  - **Statistical methods** – estimate the probability of appearing of each text symbol
    - Alphabet
    - Model
    - Coding
    - E.g: of Statistical methods: Huffman and arithmetic codin
  - **Dictionary based**
    - E.g: Ziv-Lempel family
  - **Preprocessing** for compression
    - E.g: Burrows-Wheeler

Huffman Coding is used in a variety of applications. It is a very efficient and useful lossless method of compression data. Some of its applications include: File Compression: Huffman coding is widely used in file compression algorithms (like ZIP) to reduce file sizes for storage and faster transmission.

ZIP files have always used multiple compression algorithms, but the most important are Lempel-Ziv variants and Huffman encoding.

Text compression - Statistical methods

- Defined by the combination of two tasks:
  - **Modeling** – estimates a probability for each next symbol
  - **Coding** – encodes the next symbol as a function of the probability assigned to it by the model.

Text compression – Statistical methods

Modeling

- Probabilistic Models:
  - Are widely used in IR for modeling the relevance of documents to queries. They estimate the **probability that a document is relevant to a given query** (e.g: BIM, Okapi BM25, and LMIR). These models often incorporate statistical factors such as term frequencies, document lengths, and the distribution of terms within documents and queries
- Language models:
  - Treat documents and queries as sequences of terms and model **the probability of observing a particular sequence**
  - Estimate the probability of a document given a query or vice versa, considering the **statistical properties of the language**
  - Techniques such as n-gram models, Markov models, and neural language models are used for this purpose.

Text compression – Statistical methods

Modeling

- Clustering and Classification:
  - Statistical clustering and classification methods are applied to **organize documents into groups or categories based on their statistical properties**
  - Clustering algorithms like k-means, hierarchical clustering, and Gaussian mixture models **partition documents into clusters with similar statistical profiles**
  - Classification techniques such as support vector machines (SVM), decision trees, and naive Bayes classifiers **assign documents to predefined categories based on their statistical features**

- Statistical Significance Testing:
  - Statistical significance testing is used to assess the significance of observed differences or relationships in IR experiments
  - Techniques like t-tests, chi-square tests, and analysis of variance (ANOVA) help determine whether observed differences in retrieval performance metrics are statistically significant.

Statistical methods - Modeling

Adaptative

- Are dynamic in nature and **adjust their compression strategies based on the input** data stream
- Typically use techniques such as adaptive arithmetic coding or adaptive Huffman coding
- Adaptive arithmetic coding dynamically updates the probability distributions of symbols as new symbols are encountered in the input stream
- Adaptive Huffman coding dynamically updates the Huffman tree based on the frequency of symbols encountered, allowing for variable-length encoding of symbols
- Adaptive models excel in scenarios where the statistical properties of the **input data are unpredictable or change over time**

Statistical methods - Modeling

Static

- Use **fixed compression strategies that do not change during the encoding process**
- Often employ techniques such as static Huffman coding or dictionary-based compression
- Often employ techniques such as static Huffman coding or dictionary-based compression
- Dictionary-based compression techniques, such as Lempel-Ziv-Welch (LZW), build a static dictionary of substrings encountered in the input data and encode sequences of symbols based on the dictionary entries
- Are suitable for scenarios where the statistical properties of the **input data are known in advance and remain relatively constant**

Statistical methods - Modeling

Semi-Static

- Combine aspects of both adaptive and static models by **employing adaptive techniques for certain components while keeping others static**
- Allows for efficient compression while accommodating changes in the statistical properties of the input data to some extent
- Offer a balance between adaptability and efficiency and can be **tailored to specific requirements based on the characteristics of the input** data

Lossless Compression Techniques

Variable-Length Encoding: Assigns shorter codes to more frequent terms.

Examples: Huffman coding, Arithmetic coding.

Delta Encoding: Stores differences between consecutive values instead of actual values (useful for posting lists).

Example: If posting list = [100, 105, 110], store as [100, 5, 5].

Run-Length Encoding (RLE): Compresses repeated consecutive elements.

Useful when there are long sequences of repeated terms.

Golomb Coding: Efficient for sparse data, where frequent items are close to each other.

Suitable for compressing posting lists.

Elias Gamma Coding: A universal code ideal for small, positive integers.

Often combined with inverted indexes for integer encoding.

Text compression – Statistical methods

Coding

- Compression Techniques:
  - Are employed to reduce the storage space required for large collections of text data while maintaining retrieval efficiency
  - Adaptive arithmetic coding, adaptive Huffman coding, and dictionary-based compression methods like Lempel-Ziv-Welch (LZW)
  - These methods leverage statistical properties of the data to encode text more efficiently, achieving higher compression ratio
- Error-Correction Coding:
  - Error-correction coding techniques are used to detect and correct errors that may occur during the transmission or storage of compressed text data
  - Methods such as Reed-Solomon codes, convolutional codes, and turbo codes add redundancy to the encoded data, enabling the receiver to identify and correct errors introduced during transmission or storage.

Text compression – Statistical methods

Coding

- Entropy coding:
  - Entropy coding methods exploit the statistical properties of the input data to achieve compression
  - Techniques like Huffman coding, arithmetic coding, and Golomb coding assign shorter codes to more probable symbols, reducing the average length of the encoded data.

Other Index Compression Strategies

Dictionary Compression: compresses the dictionary of terms.

- Techniques: Front-coding (store shared prefixes once), hash-based compression.

Posting List Compression: compresses the list of documents for each term.

- Blocking: Group posting lists and compress blocks separately.
- Skips and Gaps: Use skip pointers in posting lists to accelerate searches.

5. Parallel and distributed processing

Handling large-scale indexing tasks by distributing workload across multiple processors or nodes, enabling faster indexing and retrieval in massive data environments

Parallel processing

Multi-threading: Utilizes multiple threads within a single machine to process indexing tasks concurrently

- Libraries: OpenMP, POSIX Threads, Java Concurrency

SIMD (Single Instruction, Multiple Data): Executes the same operation on multiple data points simultaneously

- Useful for vector-based indexing and numerical operations.

Distributed processing

Cluster Computing: uses a network of computers (nodes) to process large datasets

- Suitable for search engines and large-scale IR systems

Distributed File Systems: provide fault tolerance and efficient data access

- HDFS (Hadoop Distributed File System): Widely used in Hadoop for large-scale data storage
- Amazon S3: Cloud-based object storage for distributed data.

Distributed processing tools

Hadoop MapReduce

- Uses MapReduce programming model for distributed indexing
- Map phase: Parses and processes documents, generating (term, document) pairs
- Reduce phase: Merges posting lists for each term
- Advantages: Fault tolerance, scalability
- Limitations: High latency for real-time indexing

Apache Spark

- In-memory distributed processing, faster than Hadoop
- Efficient for iterative tasks and real-time indexing
- Supports Resilient Distributed Datasets (RDDs) for fault tolerance.

Distributed processing tools

Elasticsearch

- Based on Apache Lucene
- Distributed, scalable, and supports real-time indexing and querying
- Uses sharding and replication for distributed data
- Optimized for full-text search with JSON-based REST APIs

Apache Solr

- Also built on Apache Lucene
- Supports sharding, replication, and advanced text search
- Effective for high-volume indexing in enterprise search systems

MongoDB

- NoSQL database with built-in text indexing
- Uses sharding for distributed processing
- Ideal for semi-structured data and high-write environments.

References

- https://users.dcc.uchile.cl/~rbaeza/mir2ed/
- https://nlp.stanford.edu/IR-book/html/htmledition/an-example-information-retrieval-problem-1.html