IR

Module 04: Indexing and Scoring in Information Systems

Inverted files, also known as inverted indexes, are a fundamental data structure used in information retrieval systems, particularly in search engines, to facilitate fast and efficient searching of large text collections.

What is an Inverted File?

An inverted file is an index structure that maps content, typically words or terms, to their locations within a set of documents. Instead of storing documents sequentially, an inverted file allows you to quickly find all the documents that contain a particular term. It essentially "inverts" the relationship between documents and terms, hence the name.

Structure of an Inverted File

An inverted file consists of two main components:

1. **Vocabulary (Dictionary):** This is a list of all unique terms (words) that appear in the document collection.
2. **Posting Lists:** For each term in the vocabulary, there is a corresponding posting list, which contains the list of documents where the term appears. Each entry in the posting list typically includes:
    - **Document ID:** Identifies the document where the term appears.
    - **Term Frequency:** The number of times the term appears in that document (optional but commonly included).
    - **Positions:** The exact positions (offsets) within the document where the term occurs (optional).

Example of an Inverted File

Let's consider a small collection of three documents:

- **Document 1:** "information retrieval is important"
- **Document 2:** "retrieval systems help find information"

- **Document 3:** "efficient retrieval of information is key"

The inverted file for these documents might look like this:

Vocabulary:

- **information**
- **retrieval**
- **is**
- **important**
- **systems**
- **help**
- **find**
- **efficient**
- **of**
- **key**

Posting Lists:

- **information:** {D1, D2, D3}
- **retrieval:** {D1, D2, D3}
- **is:** {D1, D3}
- **important:** {D1}
- **systems:** {D2}
- **help:** {D2}
- **find:** {D2}
- **efficient:** {D3}
- **of:** {D3}
- **key:** {D3}

Here's how it works:

- **Vocabulary Entry ("retrieval")** has a posting list `{D1, D2, D3}`, indicating that the term "retrieval" appears in Documents 1, 2, and 3.

- **Vocabulary Entry ("important")** has a posting list `{D1}`, indicating that "important" appears only in Document 1.

Advantages of Inverted Files

1. **Efficient Search:** By directly accessing the posting list of a term, the system can quickly identify all documents containing that term, making searches extremely fast.
2. **Space Efficiency:** While it requires additional storage compared to a simple list of documents, the space is managed efficiently, especially with techniques like compression applied to posting lists.
3. **Facilitates Boolean Queries:** Inverted files make it easy to handle Boolean queries (e.g., AND, OR, NOT operations) by intersecting or merging posting lists.

Applications of Inverted Files

- **Search Engines:** Inverted files are the backbone of search engines like Google, allowing them to quickly retrieve documents that match a user's query.
- **Text Analytics:** Used in various text analytics applications, such as sentiment analysis or document classification.
- **Database Systems:** Inverted files are also used in some database systems for indexing text fields.

Challenges and Considerations

- **Dynamic Updates:** Handling updates (insertion or deletion of documents) in an inverted file can be complex, requiring re-indexing or incremental updates.
- **Storage:** While inverted files are space-efficient, the need to store large vocabularies and posting lists for very large datasets can become a challenge.

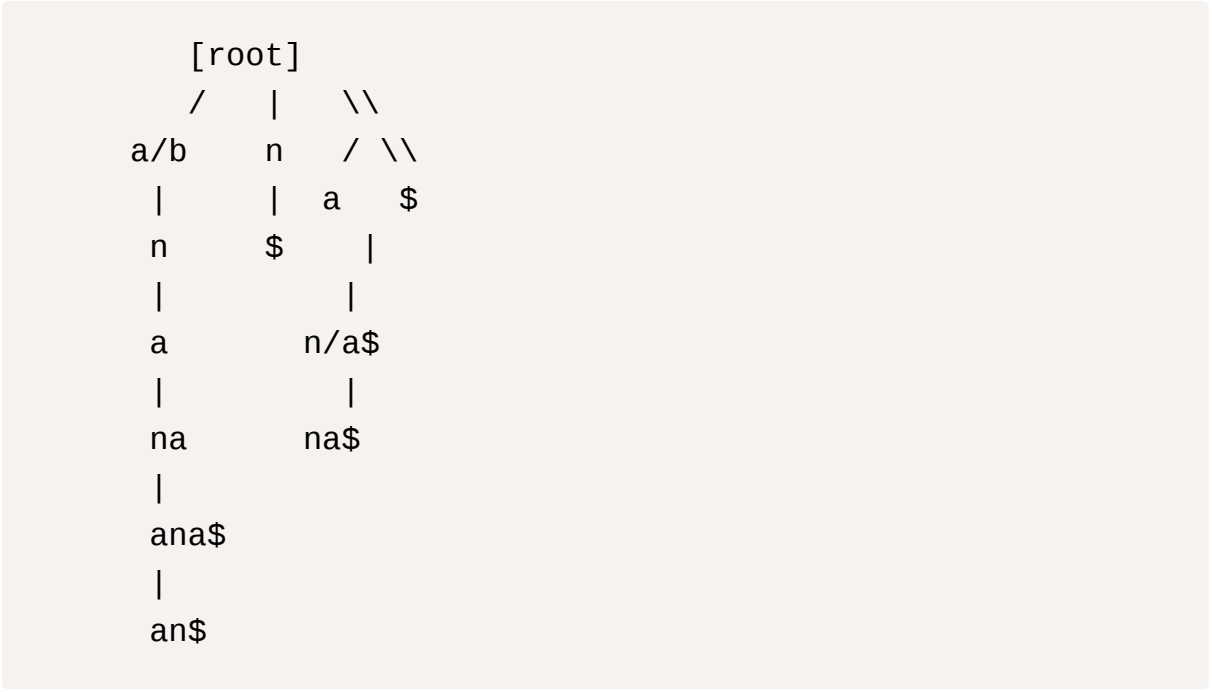
Suffix Trees and Suffix Arrays

Both suffix trees and suffix arrays are data structures used in string processing and pattern matching. They help in efficiently handling various string-related queries. Here's a brief explanation of each:

Suffix Tree

Concept:
A suffix tree is a compressed trie of all suffixes of a given string. It provides a way to store and search suffixes efficiently. Each edge in a suffix tree represents a substring of the original string, and the tree's leaves represent all the suffixes of the string.

Example:
For the string "banana", the suffix tree would look like:



- **Suffixes:** "banana", "anana", "nana", "ana", "na", "a"
- **Leaves:** Represent all suffixes of "banana".

Usage:

- Quick search for substring presence.
- Longest repeated substring.
- Pattern matching.

Suffix Array

Concept:
A suffix array is a sorted array of all suffixes of a string. It is a more space-efficient data structure compared to suffix trees and is used in conjunction with additional information, like the longest common prefix (LCP) array, to efficiently solve string problems.

Example:
For the string "banana", the suffix array is:

1. **Suffixes**:
    - "banana"
    - "anana"
    - "nana"
    - "ana"
    - "na"
    - "a"
2. **Sorted Suffixes**:
    - "a" (index 5)
    - "ana" (index 3)
    - "anana" (index 1)
    - "banana" (index 0)
    - "na" (index 4)
    - "nana" (index 2)
3. **Suffix Array**: [5, 3, 1, 0, 4, 2]

Usage:

- Pattern matching.
- Longest common prefix (LCP) computation.
- String sorting.

Differences Between Suffix Trees and Suffix Arrays

Feature	Suffix Tree	Suffix Array

Definition	A compressed trie of all suffixes of the string.	A sorted array of all suffixes of the string.
Space Complexity	Typically O(n) to O(n^2), depending on implementation.	O(n) to O(n log n) depending on sorting method.
Time Complexity (Construction)	O(n) for construction (with Ukkonen's algorithm).	O(n log n) for sorting suffixes.
Space Usage	Can be space-intensive due to additional pointers and nodes.	More space-efficient, usually requires O(n) space.
Search Efficiency	Allows for fast searches and substring queries.	Searching can be done in O(log n) with binary search.
Longest Common Prefix (LCP)	LCP is not directly available but can be computed with additional algorithms.	LCP can be efficiently computed using the suffix array.
Applications	Used for complex string operations, like finding repeated substrings, pattern matching.	Used for efficient string sorting, substring searching, and other pattern-related problems.
Complexity of Queries	More complex but can handle a variety of queries efficiently.	Simpler queries but requires auxiliary data structures like LCP array.
Construction	Construction is cheap	costly
Binary search	Binary search is not possible	Binary search is possible
	Uses Patricia tree to save memory	Uses supra index for fast retrieval
	Uses tree, non-linear data structure for construction	Uses Array, linear data structure for construction
	Uses trie as supporting data structures	Uses Arrays as supporting data structures
	Not practical for large texts	Can be used for large texts

Diagrams

Suffix Tree:

```
      [root]
       / | \\
      a  n  $
      |  |   \\
```

```
      n  a    |
      |  |    |
      a  n$  na$
      |
      ana$
```

Suffix Array:

```
String: banana
Suffixes: "banana", "anana", "nana", "ana", "na", "a"
Sorted Suffixes: "a" (index 5), "ana" (index 3), "anana" (i
ndex 1), "banana" (index 0), "na" (index 4), "nana" (index
2)
Suffix Array: [5, 3, 1, 0, 4, 2]
```

Structure of Signature Files

1. Signature file used hash function which maps words to bit masks of B bits.
2. It divides the text into blocks of 3 words each.
3. Then it assigns a bit mask of size B to each text block.
4. The mask is obtained by performing bitwise OR of the signatures of all the words in the text block.
5. So the signature file is the sequence of bit masks of all the blocks with a pointer to each block.
6. If the word is present in a text block, then all the bits set in its signature are also set in the bit mask of the text block.
7. Whenever a bit is set in the mask of the query word and not in the mask of the text block, then the word is not present in the text block.
8. Fig. 4.3.1 shows the example where the sample text is cut into blocks.
9. Block 1 Block 2 Block 3
   Block 4
   This is a text A text has many words words are Made from
   letters
   000101 110101 100100 101101 Text Signature

(text) = 000101
(many) = 110000
(words) = 001100
(made) = 001100
(letters) = 100001 Signature Function

Even if the word is not there, then also it is possible to set all the corresponding bits. And this is called as false drop.
The hash function is forced to deliver bit masks which have at least 1 bits set. A good model assumes that 1 bits are randomly set in the mask.

Searching in Signature Files

1. In signature file searching, a single word is done by hashing it to a bit mask W and then comparing the bit masks B of all the text blocks.
2. Whenever there is W and B = W (and is the bitwise AND), all the bits set in W are also set in B, and therefore the text block may contain the word.
3. So far all candidate text blocks, an online traversal must be performed to verify if the word is actually there.
4. This scheme is efficient to search phrases and reasonable proximity queries because all the words must be present in a block in order for that block to hold the phrase or the proximity query.
5. Hence, the bitwise OR of all the query masks is searched so that all their bits must be present. This reduces the probability of false drops.
6. Some case must be taken at block boundaries to avoid missing a phrase which crosses a block limit. To allow searching phrases of j words or proximities of upto j words, consecutive blocks must overlap in j words.
7. If the blocks correspond to retrieval units, simple Boolean conjunctions involving words or phrases can also be improved by forcing all the relevant words to be in the block.

Hash Addressing

Information Retrieval (MU)

- Hash Addressing (HA) is a technique to assign a location to the file using hash function on the KEY of the file.

- KEY of a file can be any unique property (like record no or name of file).
- KEY can be combination of multiple properties of a file.
- Here it is assumed that at each location there is only a single file.
- In HA, most important function is hashing function (f).
- Hashing function (f) should distribute the address of the files uniformly over the available storage.
- Hashing function (f) is bottleneck for the performance of the Scatter Storage.
- Scatter Storage fails in case of a poor hashing function (f).

Fig. 4.4.1 shows a diagram illustrating the relationship between the KEY, hashing function, and file storage.

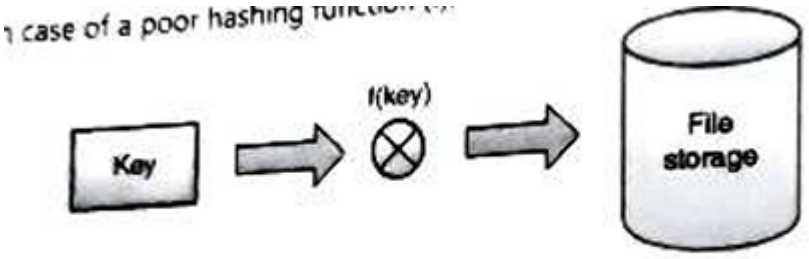


Fig. 4.4.1

Scoring

Scoring in Information Retrieval (IR) is the process of evaluating and ranking documents based on their relevance to a user's query. The goal is to determine how well each document matches the search criteria and to present the most relevant documents at the top of the search results. Here's a breakdown of how scoring works in IR:

1. Basic Concepts of Scoring

1.1 Relevance

Relevance refers to how well a document satisfies a user's query. It is typically measured by the document's ability to match the terms or concepts in the query.

1.2 Score

A score is a numerical value assigned to a document indicating its relevance to a query. Higher scores suggest greater relevance.

2. Scoring Models

Several scoring models are used to compute relevance scores. Here are some of the most common ones:

2.1 Term Frequency-Inverse Document Frequency (TF-IDF)

The TF-IDF scoring model is one of the most widely used methods in IR. It combines two components:

- **Term Frequency (TF):** Measures how often a term appears in a document. More frequent terms are considered more important.
- **Inverse Document Frequency (IDF):** Measures how rare a term is across all documents. Terms that are rare in the corpus are given higher weights.

Formula: $\text{TF-IDF}(d, t) = \text{TF}(d, t) \times \text{IDF}(t)$

Where:

- $\text{TF}(d, t)$ is the term frequency of term t in document d.
- $\text{IDF}(t)$ is the inverse document frequency of term t, computed as: $\text{IDF}(t) = \log\left(\frac{N}{\text{DF}(t)}\right)$
  Here, N is the total number of documents and $\text{DF}(t)$ is the number of documents containing term t.

2.2 Vector Space Model

In the vector space model, documents and queries are represented as vectors in a multi-dimensional space. The relevance score is computed using similarity measures like cosine similarity:

Cosine Similarity: $\text{Cosine Similarity}(q, d) = \frac{\text{TF-IDF}(q) \cdot \text{TF-IDF}(d)}{\|\text{TF-IDF}(q)\| \|\text{TF-IDF}(d)\|}$

Where:

- $\text{TF-IDF}(q)$ is the vector representation of the query.
- $\text{TF-IDF}(d)$ is the vector representation of the document.

2.3 Probabilistic Model

Probabilistic models estimate the probability that a document is relevant to a query. One common example is the Binary Independence Model (BIM), which uses probabilistic estimates to score documents.

Example: In the BM25 model, a popular probabilistic model, the score for document d give is computed as:

$$\text{BM25}(d, q) = \sum_{t \in q} \text{IDF}(t) \frac{\text{TF}(d,t) \cdot (k_1 + 1)}{\text{TF}(d,t) + k_1 \cdot (1 - b + b \cdot \frac{|d|}{\text{avgdl}})}$$

Where:

- $\text{IDF}(t)$ is the inverse document frequency of term t.
- $\text{TF}(d, t)$ is the term frequency of term t in document d.
- $|d|$ is the length of the document d.
- avgdl is the average document length in the corpus.
- k_1 and b are parameters controlling term frequency saturation and document length normalization.

3. Scoring Process Overview

1. **Document Representation:** Convert documents and queries into a suitable representation (e.g., TF-IDF vectors, term frequency counts).
2. **Score Computation:** Apply a scoring model to compute relevance scores based on the representation.
3. **Ranking:** Rank documents based on their scores. Higher scores indicate higher relevance.
4. **Result Presentation:** Present the ranked list of documents to the user.

4. Practical Example

Assume we have a query "machine learning" and two documents:

- **Doc 1:** "Introduction to machine learning"
- **Doc 2:** "The basics of programming"

For TF-IDF scoring:

1. **Calculate TF-IDF for each term in each document.**
2. **Compute the relevance score for each document based on the query terms.**

If the query terms "machine" and "learning" have high TF-IDF values in Doc 1 compared to Doc 2, Doc 1 will have a higher score and be ranked higher.

Term Weighting

Term weighting is a technique used in Information Retrieval (IR) to assign importance to terms in documents. The goal is to determine how much influence each term should have on the relevance of a document with respect to a query.

1. Term Frequency (TF)

Concept:

- Term Frequency (TF) measures how often a term appears in a document.
- The more frequently a term appears, the higher its term frequency for that document.

Formula: $\mathrm{TF}(t,d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$

Example: For the document "the cat in the hat," if the term "hat" appears 1 time in a document with 5 words: $\mathrm{TF}(\mathrm{hat},d) = \frac{1}{5} = 0.2$

2. Inverse Document Frequency (IDF)

Concept:

- Inverse Document Frequency (IDF) measures how rare a term is across all documents in the corpus.
- Terms that appear in many documents are considered less important, as they do not help in distinguishing between documents.

Formula: $\text{IDF}(t) = \log\left(\frac{N}{\text{DF}(t)}\right)$

Where:

- N = Total number of documents in the corpus.
- $\text{DF}(t)$ = Number of documents containing term t.

Example: If there are 100 documents and the term "cat" appears in 10 documents: $\text{IDF}(\text{cat}) = \log\left(\frac{100}{10}\right) = \log(10) = 1$

3. Term Frequency-Inverse Document Frequency (TF-IDF)

Concept:

- TF-IDF combines TF and IDF to give a measure of a term's importance in a document relative to the entire corpus.
- It balances the term's frequency within a document with its rarity across the corpus.
-

Formula: $\text{TF-IDF}(t, d) = \text{TF}(t, d) \times \text{IDF}(t)$

Example: Given:

- TF("hat", Doc1) = 0.2
- IDF("hat") = 1

The TF-IDF score for "hat" in Doc1: $\text{TF-IDF}(\text{hat}, \text{Doc1}) = 0.2 \times 1 = 0.2$

Term Weighting Process

1. **Compute TF for each term in each document.**
2. **Compute IDF for each term in the entire corpus.**
3. **Calculate TF-IDF for each term in each document.**
4. **Use TF-IDF scores to rank documents by their relevance to a query.**

Summary Table

Term Weighting Aspect	Term Frequency (TF)	Inverse Document Frequency (IDF)	TF-IDF
Definition	Measures term occurrence in a document.	Measures term rarity across the corpus.	Combines TF and IDF to measure term importance.
Formula			
Purpose	To highlight the importance of a term within a document.	To reduce the weight of terms that are too common.	To balance term importance and rarity.
Usage	Calculates term influence in a document.	Calculates how much a term can distinguish documents.	Determines how important a term is in a document relative to the corpus.

Formula	$\text{TF}(t,d) = \frac{\text{Count}(t,d)}{\text{Total Terms in } d}$	$\text{IDF}(t) = \log\left(\frac{N}{\text{DF}(t)}\right)$	$\text{TF-IDF}(t,d) = \text{TF}(t,d) \times \text{IDF}(t)$

Formula	$\text{TF}(t,d) = \frac{\text{Count}(t,d)}{\text{Total Terms in } d}$	$\text{IDF}(t) = \log\left(\frac{N}{\text{DF}(t)}\right)$	$\text{TF-IDF}(t,d) = \text{TF}(t,d) \times \text{IDF}(t)$

Visual Explanation

Example Document:

- Document: "the quick brown fox jumps over the lazy dog"

1. TF Calculation:

- Term "fox" appears once in a document of 9 words: $\text{TF}(\text{fox}, d) = \frac{1}{9} \approx 0.11$

2. IDF Calculation:

- Suppose "fox" appears in 5 out of 50 documents: $\text{IDF}(\text{fox}) = \log\left(\frac{50}{5}\right) = \log(10) \approx 1$

3. TF-IDF Calculation:

- Combining TF and IDF: $\text{TF-IDF}(\text{fox}, d) = 0.11 \times 1 = 0.11$

This TF-IDF score helps in ranking documents based on the relevance of the term "fox" relative to the entire corpus.