Lalitha Priya Bijja
101186225

CSC 785 - INFORMATION STORAGE AND RETRIEVAL

1.

Boolean query processing

Boolean query processing is a method of information retrieval that utilizes the Boolean operators AND, OR, and NOT to combine terms and retrieve documents that satisfy specific criteria.

In Boolean query processing, each term in the query is interpreted as a keyword, and the Boolean operators are used to specify the relationship between these keywords.

- The AND operator signifies that a document must contain all the specified terms,
- the OR operator indicates that a document must contain at least one of the specified terms.
- The NOT operator excludes documents that contain a specific term.

To process Boolean queries, an inverted index is commonly used. An inverted index is a data structure that maps terms to documents, enabling efficient lookup of documents that contain specific terms. By using an inverted index, Boolean queries can be processed efficiently, even for large collections of documents.

- Step-by-step explanation of how Boolean query processing works:

1. **Parse the query:** The query is first parsed into individual terms and operators.
2. **Identify relevant postings lists:** For each term in the query, the corresponding postings list is retrieved from the inverted index. A postings list is a list of document IDs where the term appears.
3. **Apply Boolean operators:** The postings lists are combined using the specified Boolean operators. For AND queries, the intersection of postings lists is calculated. For OR queries, the union of postings lists is calculated. For NOT queries, the postings lists are subtracted.
4. **Generate result list:** The final result list contains the document IDs that satisfy the query criteria

Example : INTERSECT operation in query

INTERSECT operation is a Boolean operator used in information retrieval.
It is used to combine two postings list and return only the documents that are common in both lists.

Brutus* AND *Caesar

Posting lists :
Brutus: 2 → 4 → 8 → 16 → 32 → 64 → 128
Caesar: 1 → 2 → 3 → 5 → 8 → 13 → 22 → 33

Iterate through the postings list
starting document ID: 2 from Brutus postings list
check if 2 is in Caesar postings list.
Yes, add 2 to the result list

Next document: 4 from Brutus postings list
check if it is in Caesar postings list
No, so move on to next doc ID in Caesar postings list
If not found then move on to the next document ID: 8 from Brutus postings list
check if its in Caesar postings list
Yes, add to the list

Repeat the process for each document ID in Brutus postings list, adding common document IDs to the result list.

***Brutus* AND *Caesar* :** 2 → 8

2.

Yes, the sorting of postings lists matters in Boolean query processing. The order of document IDs within each postings list can significantly impact the efficiency of query processing, particularly for queries involving multiple terms and Boolean operators.

The following are the reasons why sorting postings list is important:

- **Efficient Intersection Operation:** When performing an AND query, the intersection of postings lists is calculated. If the postings lists are sorted, the intersection operation can be performed efficiently using a merge sort algorithm. This approach reduces the number of comparisons required and improves the overall performance of the query.
- **Optimized Union Operation:** For OR queries, the union of postings lists is calculated. While sorting is not strictly necessary for union operations, it can still improve efficiency. Having the postings lists sorted allows for faster identification of common document IDs and reduces the number of unnecessary comparisons.
- **Efficient NOT Operation:** NOT operations involve subtracting document IDs from one postings list from another. If the postings lists are sorted, the subtraction can be performed efficiently by traversing both lists simultaneously and keeping track of the common and unique document IDs.
- **Relevance Ranking:** While Boolean query processing primarily focuses on retrieving documents that satisfy the query criteria, it can also be used to prioritize results based on relevance. Sorting postings lists by term frequency or other relevance measures allows for efficient ranking of results within the retrieved set.
- **Inverted Index Optimization**: Inverted indexes often store additional information alongside document IDs, such as term frequencies or position information. Sorting postings lists by document ID facilitates efficient access to this additional information, which can be useful for advanced ranking algorithms or proximity search.

3.

Query Optimization using Inverted Index method

Inverted Indexing :

Inverted indexing is a data structure commonly used in information retrieval systems to map terms to the documents or records that contain them. Each term in the collection has an associated postings list, which is a list of document identifiers where the term appears.

- Steps in Inverted Indexing :
  1. **Terms are Sorted:** Sorting the terms in the inverted index is a common practice. This sorting can be based on lexicographical order or other criteria. Sorting facilitates efficient search operations, making it easier to locate terms and process queries.
  2. **Duplicate Terms are Deleted:** Removing duplicate terms is an optimization that helps reduce the size of the inverted index. If the same term appears multiple times in a document, storing it only once in the index can save space and improve query processing efficiency.
  3. **Frequency Information Added:** Including frequency information in the inverted index is a crucial optimization. This information typically includes the number of times a term appears in each document (term frequency) and the total number of documents in which the term appears (document frequency). This information is vital for ranking and relevance scoring in search engines.

Example:

Consider a simple example of building an inverted index for a collection of documents:

Original Documents:

- Document 1: "cat dog fish"
- Document 2: "dog fish bird"
- Document 3: "cat bird rabbit"

Inverted Index (After Optimization):

bird: 2, 3
cat: 1, 3
dog: 1, 2
fish: 1, 2
rabbit: 3

- In this example:
  - The terms are sorted in lexicographical order.
  - Duplicate terms are removed, e.g., "dog" and "fish" appear only once in the inverted index.
  - Frequency information is added to the postings lists, indicating in which documents each term appears.

Optimization strategy for processing a Boolean query with multiple OR and AND operations :

The goal is to optimize the order in which these operations are performed to minimize the computational cost.

1. **Query Structure:**
   The Boolean query is structured as
   (X1 OR Y1) AND (X2 OR Y2) AND (X3 OR Y3), where each X and Y represents terms or conditions.
2. **Optimization Approach:** The optimization approach involves identifying pairs of OR operations within each AND operation and selecting pairs with the minimum length for early processing.
3. **Intersection with Minimum Length Pair:** After processing the minimum length pair with OR operations, the result is intersected with the pair that has the minimum length compared with the remaining pairs.
4. **Set Union Simplification:** There's a note about set union simplification, specifically $n \cup m = |n| + |m| - |n \cap m|$.
5. $n \cup m = |n| + |m|$ in a conservative way. This simplification might be applied during the union of two sets, considering only the sum of their lengths without subtracting the intersection size.

Example:

If length pair1 < length pair3 < length pair2, then
pair1 and pair3 are taken to produce intermediate results, which are then intersected with pair2.

This optimization strategy aims to minimize the number of comparisons and intersections by prioritizing the pairs with smaller lengths during the early stages of query processing.

4.

Issues in documents parsing

- **Diverse Document Formats:** Documents can be in various formats like plain text, PDF, HTML, Word, Excel, etc. Parsing documents in different formats requires specialized parsers for each type, and interoperability issues may arise.
- **Complex Structures:** Documents often have complex structures, including nested elements, tables, images, and multimedia. Parsing these structures accurately requires sophisticated algorithms and may lead to challenges in maintaining context.
- **Irregular Formatting:** Documents may exhibit irregular formatting, such as inconsistent use of fonts, spacing, or indentation. Parsing such documents accurately can be challenging, especially when trying to identify the hierarchical structure of content.
- **Language Variability:** Dealing with documents in multiple languages introduces challenges in natural language processing. Parsing may require language-specific techniques, and errors may occur when the language is not accurately identified.

- **Ambiguous Semantics:** Some documents may contain ambiguous language or semantics, making it difficult to accurately extract meaning. Resolving ambiguity often requires context-aware parsing techniques.
- **Noisy Data:** Documents may contain noise in the form of irrelevant or redundant information, typographical errors, or formatting artifacts. Noise can negatively impact the accuracy of the parsing process.
- **Incomplete Information:** Incomplete or missing information in documents can pose challenges for parsing. For example, a document might lack necessary metadata or context required for accurate interpretation.
- **Scalability:** Parsing large volumes of documents efficiently can be a scalability challenge. Efficient parsing strategies are needed to handle large datasets without consuming excessive resources.
- **Versioning Issues:** Changes in document formats or standards over time can lead to versioning issues. Parsers may need to be updated to support new formats, and legacy documents may not be parsed correctly.

5.

Tokenization -

- Tokenization is the process of breaking down a text into smaller units, called tokens.
- These tokens are typically words, phrases, symbols, or other meaningful elements that serve as the basic building blocks for further analysis in natural language processing (NLP) tasks.
- Tokenization is a crucial step in various NLP applications, including text analysis, information retrieval, and machine learning.

Process of Tokenization

1. **Sentence Tokenization:**
   The first step is often sentence tokenization, where the text is split into individual sentences. This is important because many NLP tasks operate at the sentence level.
   Example:
   Input: "This is a sentence. And this is another one."
   Output: ["This is a sentence.", "And this is another one."]

2. **Word Tokenization:**
   After sentence tokenization, each sentence is further broken down into individual words. This is the most common form of tokenization.
   Example:
   Input: "This is a sentence."
   Output: ["This", "is", "a", "sentence."]

3. **Punctuation Handling:**
   Punctuation marks are usually treated as separate tokens. However, it depends on the specific requirements of the task.
   Example:
   Input: "Hello, how are you today?"
   Output: ["Hello", ",", "how", "are", "you", "today", "?"]

4. **Handling Contractions:**
   Contractions like "don't" or "can't" are often split into two tokens: "do" and "n't", "ca" and "n't."
   Example:
   Input: "Don't worry about it."
   Output: ["Do", "n't", "worry", "about", "it", "."]

5. **Special Characters and Numbers:**
   Special characters and numbers are treated as separate tokens unless specific requirements dictate otherwise.
   Example:
   Input: "He paid $50 for that item."
   Output: ["He", "paid", "$", "50", "for", "that", "item", "."]

6. **Handling Hyphenated Words:**
   Hyphenated words may be treated as a single token or split into separate tokens based on the context or requirements.
   Example:
   Input: "high-quality product"
   Output: ["high-quality", "product"]

7. **Lowercasing:**
   Words are often converted to lowercase to ensure uniformity and to avoid treating the same word differently based on its case.
   Example:
   Input: "The Quick Brown Fox"
   Output: ["the", "quick", "brown", "fox"]

8. **Removing Stop Words:**
   Common words like "and," "the," or "is" (stop words) may be removed, depending on the task.
   Example:
   Input: "The cat is on the mat."
   Output: ["cat", "mat"]

9. **Lemmatization or Stemming:**
   Words may be further processed through lemmatization or stemming to reduce them to their base or root form.
   Example (Lemmatization):
   Input: "running"
   Output: "run"
   Example (Stemming):
   Input: "running"
   Output: "run"

10. **Named Entity Recognition (NER):**
    In some cases, named entities like names, locations, or organizations may be recognized and treated as single tokens.
    Example:
    Input: "Apple Inc. is located in Cupertino."
    Output: ["Apple Inc.", "is", "located", "in", "Cupertino", "."]

Example

Input Sentence: "The quick brown fox jumps over the lazy dog."

- Tokenization Steps:
  1. **Sentence Tokenization:**
     Output: ["The quick brown fox jumps over the lazy dog."]
  2. **Word Tokenization:**
     Output: ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog", "."]
  3. **Punctuation Handling:**
     Output: ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog", "."]
  4. **Handling Contractions:**
     Output: ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog", "."]
  5. **Special Characters and Numbers:**
     Output: ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog", "."]
  6. **Handling Hyphenated Words:**
     Output: ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog", "."]
  7. **Lowercasing:**
     Output: ["the", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog", "."]
  8. **Removing Stop Words:**
     Output: ["quick", "brown", "fox", "jumps", "lazy", "dog", "."]
  9. **Lemmatization or Stemming:**
     Output (Lemmatization): ["quick", "brown", "fox", "jump", "lazy", "dog", "."]
     Output (Stemming): ["quick", "brown", "fox", "jump", "lazi", "dog", "."]
  10. **Named Entity Recognition (NER):**
      No named entities are recognized in this simple example.

6.

Lemmatizing Text Inputs program

In [6]:
```
# Import necessary libraries
import nltk
from nltk.corpus import wordnet
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize, sent_tokenize

# Download the necessary NLTK resources
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')

# Download the WordNet corpus if not already present
nltk.download('wordnet')

# Initialize the WordNet lemmatizer
lemmatizer = WordNetLemmatizer()

# Function to lemmatize a word given its part of speech tag
def lemmatize_word(word, pos_tag):
    tag = get_wordnet_pos(pos_tag)
    if tag:
        return lemmatizer.lemmatize(word, pos=tag)
    else:
        return lemmatizer.lemmatize(word)

# Function to get WordNet POS tags
def get_wordnet_pos(treebank_tag):
    if treebank_tag.startswith('J'):
        return wordnet.ADJ
    elif treebank_tag.startswith('V'):
        return wordnet.VERB
    elif treebank_tag.startswith('N'):
        return wordnet.NOUN
    elif treebank_tag.startswith('R'):
        return wordnet.ADV
    else:
        return None
```



```
# Function to lemmatize a paragraph
def lemmatize_paragraph(paragraph):
    # Tokenize the paragraph into sentences
    sentences = sent_tokenize(paragraph)

    # Lemmatize each sentence
    lemmatized_sentences = []
    for sentence in sentences:
        # Tokenize the sentence into words and get part of speech tags
        words = word_tokenize(sentence)
        pos_tags = nltk.pos_tag(words)

        # Lemmatize each word and join them back into a sentence
        lemmatized_words = [lemmatize_word(word, pos_tag[1]) for word, pos_tag in zip(words, pos_tags)]
        lemmatized_sentence = ' '.join(lemmatized_words)

        # Add the lemmatized sentence to the list
        lemmatized_sentences.append(lemmatized_sentence)

    # Join the lemmatized sentences into a paragraph
    lemmatized_paragraph = ' '.join(lemmatized_sentences)
    return lemmatized_paragraph

# Example paragraph
input_paragraph = "Data science is a concept to unify statistics, data analysis, informatics, and their related methods to unders

# Lemmatize the paragraph
lemmatized_text = lemmatize_paragraph(input_paragraph)

# Print the original and lemmatized text
print("Original Paragraph:")
print(input_paragraph)
print("\nLemmatized Paragraph:")
print(lemmatized_text)
```

```
Original Paragraph:
Data science is a concept to unify statistics, data analysis, informatics, and their related methods to understand and analyze a
ctual phenomena with data.It uses techniques and theories drawn from many fields within the context of mathematics, statistics,
computer science, information science, and domain knowledge.However, data science is different from computer science and informa
tion science.A data scientist is a professional who creates programming code and combines it with statistical knowledge to creat
e insights from data.

Lemmatized Paragraph:
Data science be a concept to unify statistic , data analysis , informatics , and their related method to understand and analyze
actual phenomenon with data.It us technique and theory draw from many field within the context of mathematics , statistic , comp
uter science , information science , and domain knowledge.However , data science be different from computer science and informat
ion science.A data scientist be a professional who create program code and combine it with statistical knowledge to create insig
ht from data .
[nltk_data] Downloading package punkt to C:\Users\B N
[nltk_data]     Rao\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     C:\Users\B N Rao\AppData\Roaming\nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
[nltk_data] Downloading package wordnet to C:\Users\B N
[nltk_data]     Rao\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package wordnet to C:\Users\B N
[nltk_data]     Rao\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

In [2]:
```
import nltk
nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to C:\Users\B N
[nltk_data]     Rao\AppData\Roaming\nltk_data...
[nltk_data]   Unzipping tokenizers\punkt.zip.
```

Out[2]:
```
True
```

In [ ]:
```

```