PROJECT PRESENTATION

"SmartFilter: Yahoo's Advanced News Search Engine"

PRESENTED TO

Dr. Mahdi Firoozjaei

PRESENTED BY

Monika Govindaraj
Shivasree Gopinathan
Shashank Kannan
Siddharth Samber

31-07-2023

Team Members

Shashank Kannan

Web Crawler

Siddharth Samber

Hashed Trie

Shivasree Gopinathan

KMP Search

Monika Govindaraj

Edit Distance

Agenda

- Introduction
- Web Crawling and URL Preprocessing
- Building Tries & Dictionary
- Inverted Indexing for Indexing
- Hashed Trie for Searching
- Ranking
- Spellchecking and Suggestion
- KMP text search & pattern finding
- Autocompletion
- Work-Flow & Demo
- Conclusion

Introduction

Welcome to our presentation on creating a search engine!

Today, we'll be exploring the fascinating world of

- web crawling,
- indexing,
- ranking,
- autocompletion,
- spellchecking,
- and suggestions.

We'll take you through each of these functionalities step-by-step, so you can gain a comprehensive understanding of how they work and how they can be implemented in your own search engine

Search engines have become an integral part of our daily lives, helping us find information quickly and easily. But have you ever wondered how they actually work?

In this presentation, we'll delve into the technical aspects of search engines, giving you a behind-the-scenes look at the complex algorithms and processes that power them.

Web Crawling and URL Preprocessing

- Web crawling and URL preprocessing are two critical components in building an effective search engine.
- Web crawling refers to the process of automatically gathering web page data, while URL preprocessing involves cleaning and transforming URLs to make them more useful for search indexing.
- Regular expressions are commonly used in URL preprocessing to extract relevant information such as domain names, subdomains, and parameters.
- This allows search engines to better understand the content of a web page and provide more accurate search results.
- Additionally, HTML to text conversion is another important aspect of URL preprocessing, which involves converting HTML code into plain text for easier indexing and searching.

```
private final String[] excludeRegexPatterns
        "^https?://login.*",
        "^https?://ca\\.help.*",
        "^https?://ca\\.mail.*",
        "^https?://io.*",
        "^https?://*mail*",
        "^https?://mail.*",
        "^https?://help.*",
        "^https?://search.*",
        "^https?://r.*"
};
```

```java
blic void getlinks(String myURL) {

if ((!links.contains(myURL))) {
    try {
        if (links.add(myURL)) {
            int i = filteredLinks.size() + 1;
            String myString = Integer.toString(i);
            String str = myURL;
            saveUrl(myString, str);
            if (isValidLink(myURL)) {
                filteredLinks.add(myURL);
                System.out.println(i+": "+myURL);
            }else{
                j++;
            }
        }
        Document my_document = Jsoup.connect(myURL).get();
        Elements link_on_page = my_document.select("a[href]");
        for (Element page : link_on_page) {
            if (filteredLinks.size() < 25 && count < MAX_DEPTH) {
                count++;
                getlinks(page.attr("abs:href"));
            }
        }
    } catch (IOException e) {
        //System.err.println("For '" + myURL + "': " + e.getMessage()
    }
}
```

This method recursively collects links from a given URL. It avoids revisiting the same URL using a HashSet named links and checks if the link is valid according to the specified exclusion patterns using isValidLink.

This method checks if a given link is valid by comparing it against the regular expressions in excludeRegexPatterns. If the link matches any of the patterns, it is considered invalid and will be excluded from the collected links.

```java
private boolean isValidLink(String link) {
    for (String excludeRegexPattern : excludeRegexPatterns) {
        if (Pattern.matches(excludeRegexPattern, link)) {
            return false;
        }
    }
    return true;
}
```

```java
t void saveUrl(final String filename, final String urlString) {

  try
  {
      URL url = new URL(urlString);
      BufferedReader read = new BufferedReader(new InputStreamReader(url.openStream()));

      String str = filename + ".html";

      BufferedWriter write = new BufferedWriter(new FileWriter("ACC-Project-main/ACC Final Project/Acc Project/html_Files//"+

      String line;
      while((line = read.readLine()) != null)
      {
          write.write(line);
      }
      read.close();
      write.close();
  }
  catch(MalformedURLException mue)
  {
      System.out.println(x:"Malformed URL Exception");
  }
  catch(IOException ie)
  {
      //System.out.println("IOException");
  }
```

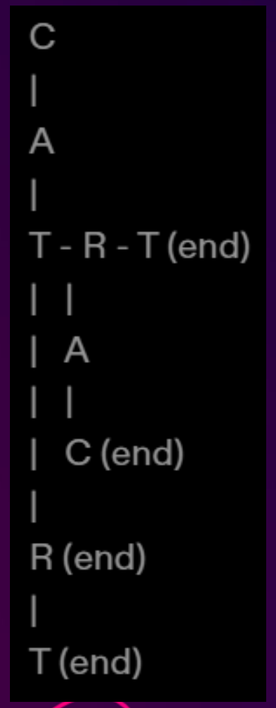
This method saves the HTML content of a URL to a file. It opens a connection to the URL, reads the content, and writes it to a file with a unique name in the specified directory (html_Files).

```java
public void runWebcrawl(){
    webCrawl crawl = new webCrawl();
    System.out.println(x:"Loading Filtered Links: ");
    crawl.PageLinks(link:"https://ca.news.yahoo.com/");
    List<String> filteredLinks = crawl.FilteredLinks();
    System.out.println("Deleted Links: "+j);
}
```

This method creates an instance of the webCrawl class, starts the web crawling process by calling PageLinks with the starting URL, and displays the number of deleted links (j) based on invalid URLs.

Building Tries & Dictionary

- A Trie, is a efficient data structure used for retrieval of keys from a set of strings. It is particularly useful for searching through large amounts of text or words.
- The Trie is constructed by starting with an empty root node, and adding each character of a string as a child node in the Trie. Each node represents a prefix of a string, and the end of a string is marked by a special flag on the node.
- For example, consider the words 'cat', 'car', and 'cart'.



- A dictionary is a powerful tool used in various applications for organizing, storing, and retrieving words efficiently.
- The process of building the dictionary involves iterating through the words in all the text files while simultaneously creating the trie data structure. The words are then appended to a text file named "Dictionary.txt."
- Trie Indexing creates a tree-like structure that represents all possible prefixes of the words in the database, allowing for quick lookups based on partial matches.
- One of the main benefits of Trie Indexing is its speed.
- Additionally, Trie Indexing is very memory-efficient, as it only stores the necessary information and discards the rest

Searching, Indexing, and Ranking

- The process of searching, indexing, and ranking in a search engine is a complex one that requires careful consideration of various factors.

- One important technique for efficient indexing and searching is the use of hash trie data structure. This allows for fast lookups and retrieval of information, which is crucial for a search engine to operate smoothly.

- When it comes to ranking algorithms, there are several options available, each with its own strengths and weaknesses. Some popular examples include PageRank, TF-IDF, and BM25.

- These algorithms take into account various factors such as keyword frequency, link popularity, and relevance to determine the most relevant results for a given query.

Hashed Trie and Inverted Indexing

```java
class TrieNode {
    private Map<Character, TrieNode> children;
    private boolean isEndOfWord;
    private Map<Integer,Integer> freq_table;
    private int user_searched;

    public TrieNode() {
        children = new HashMap<>();
        freq_table=new HashMap<>();
        user_searched=0;
    }
    // Tracking User searches
    public int  get_search_times() {
        return user_searched;
    }
    public void update_search_time() {
        user_searched=user_searched+1;
    }
    // Tracking trie children
    public Map<Character, TrieNode> getChildren() {
        return children;
    }
    // Tracking freq_table
    public void setfreqt(int x,int y) {
        freq_table.put(x, y);
    }
    public int  getfreqt(int x) {
        if(freq_table.get(x)==null)
        {
            return 0;
            // return 0 if table has no entry for x
        }
        return freq_table.get(x);
        // else it returns the corresponding value
    }
    public  Map<Integer,Integer> get_full_table() {
        return freq_table;
        // it return whole table or hashmap , it could be empty or filled
    }
}
```

Trie Node consist

Children : HashMap which consist Character and trieNode mapping

Freq_table : HashMap which consist File number and frequency of word in that file mapping

User_searched : it represents the number of times user searched a particular word . Could be useful for tracking History.

It consist of different getters and setters for all these Parameters

- Using Hashed trie saves memory
- It also allows for dynamic growing and shrinking of trie .
- It makes retrieval of particular trie node faster using hashing.

Search & Insert

Insert

- Takes input the file number and word
- For each character of word it inserts to its children if absent
- It updates current to the new children created if absent.
- It traverse till the last character in same manner
- It set file number and value 1 in frequency map
- It set file number and increment value in frequency map if file number key already exists

Search

•It takes the word as input and returns the frequency map and if not found then it returns null
•For each character of word it checks if character is there in map
•It updates current to the new children
•If children becomes null during this process it returns null
•Else returns the frequency map

```
public Map<Integer,Integer> search(String word)
    TrieNode current = root;
    for (char ch : word.toCharArray()) {
        current = current.getChildren().get(ch);
        if (current == null) {
            return null;
        }
    }
        current.update_search_time();
        return current.get_full_table();
}
```

```
public void insert(String word,int file_number) {
    TrieNode current = root;
    for (char ch : word.toCharArray()) {
        current.getChildren().putIfAbsent(ch, new TrieNode());
        current = current.getChildren().get(ch);
    }
    if(current.getfreqt(file_number)==0)
    {
        // 0 means that there there is no entry
        current.setfreqt(file_number, 1);
    }
    else
    {
        int temp =current.getfreqt(file_number)+1;
        current.setfreqt(file_number,temp);
    }
}
```

Ranking using Max-Heap

- A max Heap has been built using Priority Queue.
- Priority Queue stores entery map of 2 integers depicting (File number , Number of repetition of the word)
- Entries are sorted based on which entry has largest number of repitions.
- It returns the string of filename which has maximum frequency and print the frequency map in decsending order of frequency

```java
public static String priority( Map<Integer,Integer> freq_table) {

    // Create Comparator Descending order
    Comparator<Map.Entry<Integer, Integer>> valueComparator = (entry1, entry2) -> entry2.getValue() - entry1.getValue();

    //Initialize a PriorityQueue with the custom comparator
    PriorityQueue<Map.Entry<Integer, Integer>> maxHeap = new PriorityQueue<>(valueComparator);


    //Add each key-value pair to the max-heap
    if(freq_table!=null)
    {
        if(freq_table.isEmpty())
        {
            return null;
        }
        else
        {
            maxHeap.addAll(freq_table.entrySet());
        }
    }
    else if(freq_table==null)
    {
            return null;
    }



    //Retrieve elements from max heap
    max=0;
    while (!maxHeap.isEmpty()) {
        Map.Entry<Integer, Integer> entry = maxHeap.poll();
        int key = entry.getKey();
        int value = entry.getValue();
        if(value>=max){max=value;k=key+".txt";}
        else if(value<max){}
        System.out.println("File Number: " + key + ", Frequency: " + value);
    }
    return k;
}
```

Spellchecking and Suggestion - Edit distance

- Spellchecking and suggestion are crucial functionalities in a search engine.
- In today's fast-paced world, users expect search engines to not only provide relevant results but also offer suggestions for misspelled words.
- Spellchecking ensures that the user's query is correctly understood, while suggestion helps in providing alternative options if the query does not yield desired results.
- Edit distance is a popular algorithm used for word suggestion in search engines.
- It calculates the minimum number of operations required to transform one word into another.
- This algorithm can be used to suggest alternate spellings of a word or offer related terms based on the user's input.
- Implementing these functionalities in a search engine can greatly improve the user experience and increase user engagement.

```java
public class EditDistance {

    public static int minDistance(String word1, String word2) {
        int m = word1.length();
        int n = word2.length();

        int[][] cost = new int[m + 1][n + 1];
        for(int i = 0; i <= m; i++)
            cost[i][0] = i;
        for(int i = 1; i <= n; i++)
            cost[0][i] = i;

        for(int i = 0; i < m; i++) {
            for(int j = 0; j < n; j++) {
                if(word1.charAt(i) == word2.charAt(j))
                    cost[i + 1][j + 1] = cost[i][j];
                else {
                    int a = cost[i][j];
                    int b = cost[i][j + 1];
                    int c = cost[i + 1][j];
                    cost[i + 1][j + 1] = a < b ? (a < c ? a : c) : (b < c ? b : c);
                    cost[i + 1][j + 1]++;
                }
            }
        }
        return cost[m][n];
    }
}
```

The minDistance function in the provided code calculates the minimum edit distance between two input strings word1 and word2. The edit distance is the minimum number of operations (insertion, deletion, or substitution) needed to convert word1 to word2. It uses a dynamic programming approach, initializing a 2D array cost to store intermediate results. The algorithm iterates through the characters of both words, updating the cost array based on matching and non-matching characters. Finally, it returns the minimum edit distance stored at cost[m][n], where m and n are the lengths of word1 and word2, respectively.

KMP Search

Autocompletion is a feature that can greatly improve the user experience in a search engine. By predicting what the user is typing, it can save them time and effort, and make their search more efficient.

- This functionality can be implemented using various algorithms, such as trie or ternary search tree.
- By storing all possible words and phrases in a data structure, the search engine can quickly retrieve suggestions based on the user's input.
- KMP search, on the other hand, is a string-matching algorithm that can be used for keyword searching in files.

Working:

It works by finding the occurrence of a pattern within a larger text, and can be useful in searching large databases or documents.

By pre-processing the pattern to find the longest proper suffix that is also a prefix of the pattern, KMP search can achieve linear time complexity, making it an efficient solution for keyword searching.

Knuth-Morris-Pratt (KMP) Algorithm for Text Search

The Knuth-Morris-Pratt (KMP) algorithm efficiently matches patterns in text. The algorithm compares the search word and the text character by character.

It uses a Longest Prefix Suffix (LPS) array for pattern optimization.
If a mismatch occurs, the algorithm adjusts the search position using LPS values, avoiding redundant comparisons. Case-insensitive search is supported.

Advantages:

- Fast pattern matching for large text files.
- It handles separate word detection within the text.

Example:
File: "example.txt"
Search Word: "goal"
Output: Occurrences of "goal" in the file: example.txt are found at positions: [47, 125, 248].

KMP Algorithm: Finding Patterns Efficiently

Initialization: Compute the Longest Prefix Suffix (LPS) array for the pattern

- Here a string matching algorithm that searches for occurrences of a pattern within a larger text.
- The LPS array is computed for the pattern and is used to optimize the search process by avoiding unnecessary comparisons between characters in the text and pattern.
- To construct the LPS array, we initialize the first value as 0 since there are no proper prefixes of length greater than or equal to 1 that are also suffixes. We then iterate through the rest of the pattern array and compare the characters at the current and previous indices. If they match, we increment the value of the LPS array by 1.

```
int M = pattern.length();
int N = text.length();

int[] lps = computeLPSArray(pattern);

int i = 0; // index for text[]
int j = 0; // index for pattern[]

while (i < N) {
    if (pattern.charAt(j) == text.charAt(i)) {
        i++;
        j++;
    }

    if (j == M) {
        // Check if it is a separate word
        if (isSeparateWord(text, i - j - 1, i)) {
            occurrences.add(i - j);
        }
        j = lps[j - 1];
    } else if (i < N && pattern.charAt(j) != text.charAt(i)) {
        if (j != 0) {
            j = lps[j - 1];
        } else {
            i++;
        }
    }
}

return occurrences;
```

- If not, we backtrack to the previous index and check if the character at the new index matches the current character. This process continues until we reach the end of the pattern array.

Example, consider the pattern 'ababc'.

- The LPS array for this pattern would be [0, 0, 1, 2, 0]. At index 2, we have 'a' and 'b', which do not match. We backtrack to index 1 and compare 'a' and 'a', which match.
- Therefore, the value at index 2 of the LPS array is 1. At index 3, we have 'a' and 'b', so we backtrack to index 2 and compare 'b' and 'b', which match.
- Therefore, the value at index 3 of the LPS array is 2. Finally, at index 4, we have 'c' and 'b', so we backtrack to index 2 and compare 'b' and 'a', which do not match. We then backtrack to index 1 and compare 'a' and 'c', which also do not match. Therefore, the value at index 4 of the LPS array is 0

Search: Iterate through the text and pattern to find occurrences.

- It starts by comparing the first character of the pattern with the first character of the text. If they match, it moves on to compare the second character of the pattern with the second character of the text, and so on.

Occurrence Handling , Separate Word Check & Time Complexity

- KMP uses the LPS array to skip over previously matched characters and continue searching from the next possible match.
- This approach ensures that we do not miss any occurrences of the pattern in the text and improves the overall efficiency of the algorithm.
- Complexity: KMP algorithm has a time complexity of O(N + M), where N is the length of the text and M is the length of the pattern.

```java
private static int[] computeLPSArray(String pattern) {
    int M = pattern.length();
    int[] lps = new int[M];
    int len = 0; // length of the previous longest prefix suff

    lps[0] = 0;
    int i = 1;

    while (i < M) {
        if (pattern.charAt(i) == pattern.charAt(len)) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }
    return lps;
```

```java
public static void runKMPsearch(String fz, String x) {
    String fileName = "text_Files//" + fz; // Replace with the path to your te
    String searchWord = x; // Replace with the word you want to search

    List<Integer> occurrences = searchWordInFile(fileName, searchWord);
    System.out.println("By using KMPSearch the Occurrences of '" + searchWord
}
```

Autocompletion using DFS

```
public TrieNode search2(String word) {
    TrieNode current = root;
    for (char ch : word.toCharArray()) {
        current = current.getChildren().get(ch);
        if (current == null) {
            return null;
        }
    }
        return current;
}
```

Search2
- It traverse the tire using the searched word or prefix
- If null is returned , it means there is no such prefix
- Else it returns TrieNode associated with last character

```
public List<String> autocomplete(String word) {
    List<String> suggestions = new ArrayList<>();
    TrieNode node = search2(word);
    if (node != null) {
        collectSuggestions(node, word, suggestions);
    }
    return suggestions;
}
```

Auto-compelte
- It has suggestion List which will store the words with certain prefix
- If prefix exist in trie it calls collect suggestion method

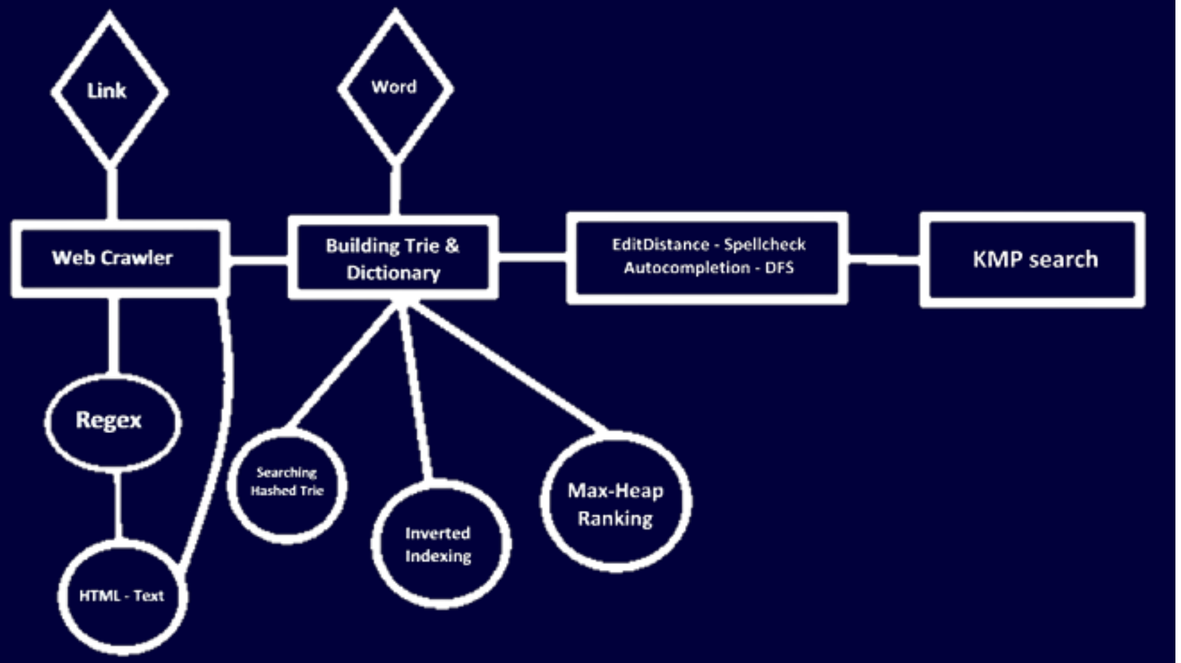
```
private void collectSuggestions(TrieNode node, String prefix, List<String> suggestions) {

    if (!node.get_full_table().isEmpty()) {
        suggestions.add(prefix);
    }
    for (Map.Entry<Character, TrieNode> entry : node.getChildren().entrySet()) {
        Character key = entry.getKey();
        TrieNode value = entry.getValue();
        if(value!=null)
            collectSuggestions(value, prefix + key, suggestions);
    }

    }
```

For each entry in map of the trieNode we traverse depth wise and keep on adding characters to make up the word for suggestion

WORKFLOW



DEMO

```
PS C:\Users\Shashank\Desktop\latest_acc\ACC-Project-main>  c:; cd 'c:\Users\Shashank\Deskto
les\Java\jdk-15.0.2\bin\java.exe' '-agentlib:jdwp=transport=dt_socket,server=n,suspend=y,ad
ta\Local\Temp\cp_4y3g0jlpyujyth8jt84jhmqno.argfile' 'project.indexMain'
Loading Filtered Links:
```

Debug 'in

Source: Debug

CONCLUSION

- We started with web crawling and URL preprocessing, which are critical for extracting relevant information from the web.
- Then discussed searching, indexing, and ranking, where hash trie data structure can be used for efficient indexing and searching.
- Additionally, we explored autocompletion and KMP search, which can enhance user experience in a search engine. Finally, we talked about spellchecking and suggestion, which are crucial for providing accurate results to users.
- These techniques are essential for efficient and effective search operations in various applications.
- By understanding these concepts, you can improve the performance of your search engine or application and provide better user experience to your customers.
- It is important to keep up with the latest advancements in search technologies and implement them in your projects to stay ahead of the competition.