Joojle Search Engine

Introduction

In this final project you will build a search engine for one site. The search engine will accept search queries and will respond with a list of links to pages containing the search terms. The list will be ordered by importance [ see later for details].

For testing we will be using the site: https://harrypotter.fandom.com/wiki/Main Page

But you can use any internal or publicly available site.

High Level Requirements

The system will have the following requirements.

a. Crawling and link discovery subsystem.

b. Search engine for executing queries.

c. Ranking subsystem to assign importance to pages.

Crawl One Site

1. The system will crawl one site given by a start URL in one of the configuration files.

2. The site will can be crawled either in breadth first or a depth first manner.

The mode used should be set in configuration.

See Wikipedia for details:

a. https://en.wikipedia.org/wiki/Breadth-first search

b. https://en.wikipedia.org/wiki/Depth-first search

3. We can set a max depth for the crawling in the configuration.

a. If a max depth is set, do not crawl more deeply.

b. If no max depth is set, then crawl until you process all pages in the site that are

reachable from the start **URL**.

4. We can set a max number of pages to be visited. If not configured or configured to 0 then the

number of pages is not bounded.

5. The system will expose a search RESTful API that will allow execution of search queries.

a. Execute Search queries.

b. Obtain statistics about quantities of executed search queries:

i. How many

ii. List of executed queries and number of results for each

6. The system will expose a statistics RESTful API that will allow for obtaining live information

about the crawling process:

a. Is crawling still in progress or finished.

b. How many pages have been visited so far and how many links where discovered. c. Max depth reached.

**1**

Search Engine

The search engine will be exposed by a RESTful API. It will support executing search queries having one or more positive terms and optionally one or more negative terms.

a.

Positive term

i. One word starting with a letter, minimum length is 3.

ii. Result will be pages containing the term.

b. Negative term

I. One word beginning with a minus sign.

ii. Result will be pages not containing the term.

Query Requirements:

1. Search words are case insensitive.

2. Search for one positive term will return a list of pages that contain the term:

Expelliarmus

3. Search for two or more positive terms will return a list of pages that each contain all of terms:

Petrificus Totalus

4. Search for pages with a one or positive more terms and one or more negative terms. The result returned is a list of pages that contains all of the positive terms but none of the negative terms. Examples:

**0.**

dark -matter

this will return all pages with "dark" but without "matter"

b. dark lord*-*matter-harry

this will return all pages with "dark" and "lord" but without "matter" or "harry"

5. Search query with only negative terms is not allowed.

2

© 2023 Muhammad Zahalqa m@tryfinally.com

Implementation Hints

Implementation should be done using:

1. Maven reactor model.

2. Spring Boot.

3. Any required external dependency to fetch, parse and process HTML pages and links.

Configurations

The system will at least have the following configurations, taken either from a config file or command line arguments.

URL link or links to start the crawl from.

Maximum number limit of pages to crawl or unlimited.

• Maximum depth to reach or unlimited.

Crawl mode: BFS or DFS

Web Crawler

The crawler will start the crawling process from a set of at least one URL (from configuration). Usually this will be the main page of the site.

The crawler will download the page and extract all links from it. Links are present on the page in the form of <a href="link" /a> html tags. The crawler will:

1. Extract all links from this page and insert them into a data structure of new links that need to be

visited.

2. Send the page text to an indexer component to update the index.

3. Repeat for all links in the data structure holding unvisited links.

4. Care must be taken not to crawl the same link more than once.

Design Highlights

Initially the crawler will not be multithreaded. But in later phase you should enhance it to be multithreaded. For that it's essential you start you implementation using thread safe data structures.

Result of crawling

The result of the crawling is to produce:

a map of links between the pages

for each visited link URL, store the number of links to any page from the site that is linked from this.

an inverted index of words indicating which pages contains which words.

Work Plan

Project directory: joojle

Each mile stone should contain a working project and will be developed in its own branch. Name the branches: joojle-mx where x is the milestone number. Like:

1. joojle-m1

2. joojle-m2

Milestone 1

Implement a search engine that has the following phases:

**•** Crawl: Server will crawl the web site/s taken from the configuration file, when this phase is done

it will print a message with the count of unique pages visited and the total number of pages indexed on the log.

Once this is done, user can execute queries. The queries will be handled on the command line. (No RESTful APIs yet)

Search: Prompt the user to enter search queries and responds with search results.

• Search queries should be one word   
  
  
Milestone 1

Implement a search engine that has the following phases:

• Crawling Phase

o Crawl: Server will crawl the web site/s taken from the configuration file

o When this phase is done it will print a message with the following information:

■ count of unique pages visited.

■ Total number of links encountered discovered. (Counting duplicates)

■ Total number of links on visited pages which get ignored as they link to site outside of the crawl group.

• Query Phase

o Once the crawling phase is finished, the user can execute queries.

o Prompt the user on the console to enter a search query and respond with results.

o Search queries should be one term.

o Search results will be sorted according to "importance."

■ Page importance will be measured by the number of times the search term appears in the page.

o Results will be displayed as:

■ Query executed

■ List of lines one for each result

• Page title. Taken from the page <t it le> tag if present.

• Page's Link.

• Number of times the search word appears in the page.

Milestone 2

Search results will sorted according to "importance"

Page importance will be measured by the number of times the search term appear in the page.

Add handling of multiple search words. For instance, searching for cat dog will return only pages having

both words**.**

Milestone 3

Add handling of multiple search words where some words are negative. For instance, searching for cat *dog-*parrot will return only pages having both cat and dog but not parrot.

search words prefixed with - must not be present on any of the results returned.

Milestone 4

Implement a search server that can be queried over RESTful APIs as defined above.

Milestone 5

Add multithreading support to the crawler. The crawler will be executed by multiple. This will mandate the use of thread safe data structures.

The RESTful APIs will be available even before the crawler finished its work.

Milestone 6

Implement PageRank based page importance for sorting search result. The PageRank algorithm is described in

https://en.wikipedia.org/wiki/PageRank

https://youtu.be/RVlr8YSisek

*simplified explanation:*

Page rank algorithm simulate a random surfer that surfs between the pages. The algorithm calculates the probability of ultimately being on a page. Assume we have the following pages that link to each other:

We say a page is important if it's connected with important pages. We will use a random walk simulating a surfer that clicks randomly on the links. So, if a surfer is on page 2, he has 50% chance of surfing to page 1 and 50% to surf to page 2.

Let's start by giving each page an equal rank of 1.0. Now, each page will divide its rank among all the pages it links to. We will describe this using transposed, column-normalized matrix - transition matrix. Columns describe the source pages and rows describe the destination pages. When the surfer transition to a page, part of the rank of the source page will flow to the destination.

as page 0 links to 3 pages, a surfer has 1/3 probability to leave it to one of these pages (1st column) as page 3 links only to page 1, a surfer will move to that page with probability 1.0. (4th column)

5

Now the surfer will continue the random surfing by flowing the links and the ranks will flow to the more important pages. We need to find the steady state of this flow.

We can either use linear algebra (see Wikipedia article and YouTube link) or using an iterative process that will eventually converge (mostly):

How do calculate the steady state iteratively:

1. Use a hash table to store the ranks of all pages. Such that rank[page] will be initialized to 1.0 2. Compute the flow of ranks

Rank of page i in the next phase is equal to all the ranks contributed by all pages linking to it.

3. After computing the page rank at time t+1 we iterate the computation until we reach a steady state or reach maximum of T iterations. (taken from configuration or command line, "80).

Now that we have the page ranks, we can calculate a weighted importance score for each page using the PageRank of the page and the number of times the search word appears in in it.

Sort the search results using the combined score calculated as PageRank \* Occurrences.

For instance, when searching for *Harry* the combined score of the page: https://harrypotter.fandom.com/wiki/Troll is computed using PR \* S

Milestone 7

Implement a save and load mechanism to store the calculated index and page ranks to a database. What database will you select to use?

And implement **a** mechanism for the search engine server to start serving search queries using the stored data.

6

© 2023 Muhammad Zahalqa

m@tryfinally.com

Milestone 8

Enable the client to execute search queries before we finish executing and finishing the crawl process. To make this usable we need to recalculate the PageRank every n URLs discovered.

You need to lock the data structures created in previous phases and merge the new discovered data with the existing data from previous step and recalculate the PageRanks.

Make the server prints a message when this happens.

As the process continues, we will **see** that queries might return different results now as the crawler will keep updating the index and the ranks. 7