NetNinjas Final Report

A Web-based Search Engine Project

Introduction

NetNinjas is a web-based search engine that strives to give users quick results that are also accurate. The idea of web crawling, processing, ranking, and user interface/frontend design is the foundation of the project. The different facets of the project development process, such as the technical implementation, features, difficulties, and prospective future improvements, will be included in this report.

Features

Crawling

- Recovery
    - The crawler stores URLs in a table immediately when new URLs are extracted from the page content. All URLs which are not crawled can be reloaded from the table when the crawler starts running.
- Fault tolerance and logging
    - All exceptions thrown will be cached during the crawling process and logged with a timestamp. There are time limits for connections (1 second) and crawling (2 seconds).
- Finding canonical and foreign-language web pages
    - The crawler identifies whether a page is a canonical or foreign-language page by checking information stored in link tags and HTML tags. Additionally, the percentage of English characters on a page is also used to determine if the content is in English.
- Crawling with priority
    - Failure to crawl a page will be recorded and used as a judgment for crawling priority later.
    - Only links to non-similar hosts will be put in the URL queue.
    - If there are too many links to a single host, most links will be dropped.
- Removing useless content
    - Most HTML tags are removed. For robots.txt, only User-agent: stored * part.

Processing

- Parse HTML page with regex
    - Delete all tags and scripts using regex
    - Extract the title in the “head” tag
    - Save a plain page table for page preview
- Generate the index table with TF and IDF precomputed
    - Precompute and save TF and IDF in the index table for ranking

    - Filter single characters and words that are too long
- Generate the PageRank table, using a general algorithm.
- Optimize for large data
    - Add cleaning methods in flame. Delete useless RDDs to save storage.
    - Distribute pairRDD equally to each node to improve efficiency by concatenating part of the hash result before the key of the pair.
    - Restartable jobs and continue previous progress after unexpected interruption.

Ranking

- Search Word Processing
    - tokenization -> normalization -> stop word removal -> word stem
- Synonyms Finding
    - Obtain one word's vector representation from the GloVe and calculate its cosine similarity with other words.
- Search suggestion
    - Use the trie tree data structure to take a prefix string and return a list of suggested search terms.
- Search history caching: Cache the latest searched page.
- TF-IDF
    - Phrase search: retrieve two words' positions from the index table and calculate the distance between them. If they are close enough, then we regard them as phrases and give them higher weights.
    - Single root word search: retrieve the TF-IDF of each search word.
    - Synonym word search: retrieve the TF-IDF of each word's synonym.
- Priority Score with Pagerank: combine TF-IDF, PageRank, and title scores.
- Garbage collection: GC the search results every 30 minutes.

Frontend

- Infinity Scrolling
    - Using Infinity Scrolling eliminates the need to manually load new pages or press pagination buttons in order to scroll down an endless number of search results.
- Autocomplete
    - Based on the user's input, this feature asynchronously retrieves and filters a list of suggested search terms from the server using JavaScript and AJAX.
- Past Search
    - The previous search queries matching the current input are shown in a dropdown list when the user clicks on the search field on the index page. The list of previous searches is presented in a user-friendly manner, placing the most recent search requests at the top.
- Route guard
    - To ensure that the user is directed to the index page if they attempt to navigate to a search results page without typing a query into the search bar, a route guard was developed at the front end of this project.

Challenges & Difficulties

Challenges

Large url queue

The URL queue grows very fast during crawling and it's not possible to crawl all URLs. To solve this problem, we collect some data for different hosts during the crawl. The number of occurrences of non-English content bobots.txt restriction, and failed access will affect the crawling decision. Moreover, if there are too many URLs in the queue for a single host, we won't add any more URLs for that host.

Out of memory

When the number of crawling pages is large, it's easy to encounter the problem of being out of memory in the processing process. Our solutions are: to process data in multiple batches, delete intermediate tables in time, and make some large tables persistent.

Phrase Search

Hard to identify whether two or more words are phrases or not. Our solution is: if the words are all on the same page, then compare their positions in the index table to determine how close they are to each other. If they are close enough, then we regard them as phrases.

Most Difficult Aspects of the Project

- Determining the crawling priority of the URLs is quite difficult. The quality of a URL depends on a lot of factors and there isn't really a general method to determine whether a URL is good or bad.
- Handling black-hat SEO is difficult. There are various types of black-hat SEO, and it's really hard to avoid all of them. Sometimes web pages with low quality (e.g. many links pointing to different hosts with similar contents) would appear in our search results.
- During the deployment process, a number of difficulties were encountered, including compatibility problems with the server environment, issues with the installation and setup of dependencies, and problems configuring the database and connecting it to the backend code.

What would we do differently?

- Modify and optimize the "flame" code for large amounts of data at first, including providing options for RDD to be persistent, changing the storage format of the "pairRDD" in KVS table so that it's always distributed equally among workers, etc.
- Test on EC2 early to find out differences between local and cloud, so that we could make targeted changes and optimization early.