Back To Module Home

Design Problems

0% completed

RESHADED Approach for System Design

Design YouTube

Design Quora

Design Google Maps

Design a Proximity Service / Yelp

Design Uber

Design Twitter

Design Newsfeed System

Design Instagram

Design a URL Shortening Service / TinyURL

Design a Web Crawler

Design WhatsApp

Design Typeahead Suggestion

Design a Collaborative Document Editing Service / Google Docs

Conclusion

Mark Module as Completed

Design Improvements of a Web Crawler

Identify the web crawler's design shortcomings and challenges, and make improvements accordingly.

We'll cover the following

Introduction#

This lesson gives us a detailed rundown of the improvements that are required to enhance the functionality, performance, and security of our web crawler design. We have divided this lesson into two sections:

1. Functionality and performance enhancement design improvements—extensibility and multi-worker architecture.
2. Security-enhancement design improvements—crawler traps.

Let’s dive into these sections.

Design improvements#

Our current design is simplistic and has some inherent shortcomings and challenges. Let’s highlight them one by one and make some adjustments to our design along the way.

- **Shortcoming**: Currently, our design supports the HTTP protocol and only extracts textual content. This leads to the question of how we can extend our crawler to facilitate multiple communication protocols and extract various types of files.

  Adjustment: Since we have two separate components for serving communication handling and extracting, HTML Fetcher and Extractor, let's discuss their modifications one by one.

  1. **HTML Fetcher**: We have only discussed the HTTP module in this component so far because of the widely-used HTTP URLs scheme. We can easily extend our design to incorporate other communication protocols like File Transfer Protocol (FTP). The workflow will then have an intermediary step where the crawler invokes the concerned communication module based on the URL's scheme. The subsequent steps will remain the same.
  2. **Extractor**: Currently, we only extract the textual content from the downloaded document placed in the Document Input Stream (DIS). This document contains other file types as well, for example, images and videos. If we wish to extract other content from the stored document, we need to add new modules with functionalities to process those media types. Since we use a blob store for content storage, storing the newly-extracted content comprising text, images, and videos won't be a problem.

The extensibility of the HTML fetcher and extractor

- **Shortcoming**: The current design doesn't explain how the multi-worker concept integrates into this system.

  Adjustment: Every worker needs a URL to work on from the priority queue.

Different websites require different times for workers to finish crawling, so each worker will dequeue a new URL upon its availability.

There are several ways to achieve this multi-worker architecture for our system. Some of them are as follows:

1. **Domain level log**: The crawler assigns one whole domain to a worker. All the URLs branching out of the initial URL are the responsibility of the same crawler. We ensure this by caching the hash of the hostname against the worker ID for guaranteed future assignment to the same worker. This also helps us avoid any redundant crawling of that domain's web pages by any worker.

   This approach is best-suited for achieving **reverse URL indexing**, which involves traversing the web address directories in a reversed order. It ensures the storage efficiency of the URL storing process for later use and prevents extensive repetitive string matching processes for duplication testing.

2. **Range division**: The range of URLs given to each crawler; the crawler distributes a range of URLs from the priority queue among workers to avoid clashes. Like the previous approach, the crawler must hash the range associated with each worker.

3. **Per URL crawling**: Queue all the URLs; a worker takes a URL and enqueues the subsequently found URLs into the priority queue. Those new URLs are readily available to crawl through for other workers. To avoid enqueuing multiple similar links that direct to the same web page, we calculate the checksum of the canonicalized URL.

Crawler traps#

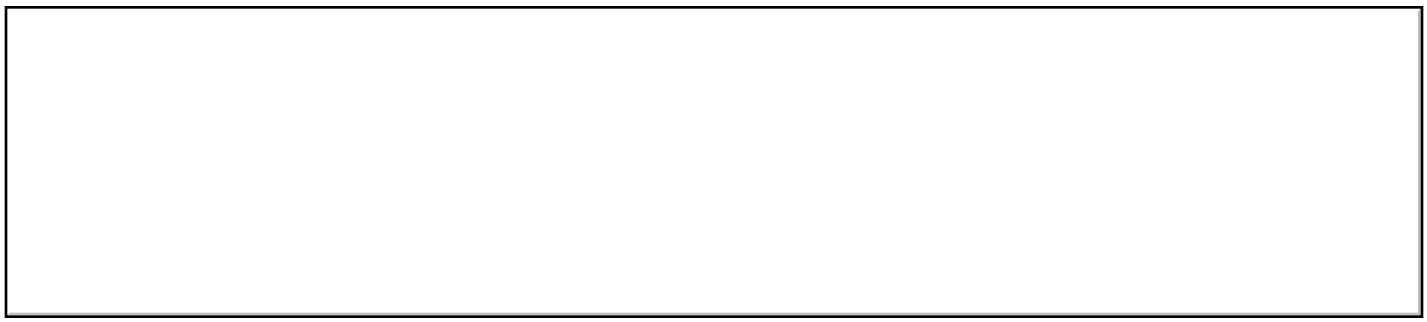
A **crawler trap** is a URL or a set of URLs that cause indefinite crawler resource exhaustion. This section dedicates itself to the classification, identification, and prevention of crawler traps.

Classification#

There can be many classification schemes for crawler traps, but let's classify them based on the URL scheme.

Mostly, web crawler traps are the result of poor website structuring, such as:

- **URLs with query parameters**: These query parameters can hold an immense amount of values, all while generating a large number of useless web pages for a single domain: `HTTP://www.abc.com?query`.
- **URLs with internal links**: These links redirect in the same domain and can create an infinite cycle of redirection between the web pages of a single domain, making the crawler crawl the same content over and over again.
- **URLs with infinite calendar pages**: These have never-ending combinations of web pages based on the varying date values, and can create a large number of pointless web pages for a single domain.
- **URLs for the dynamic content generation**: These are query-based and can generate a massive number of web pages based on the dynamic content resulting from these queries. Such URLs might become a never-ending crawl on a single domain.
- **URLs with repeated/cyclic directories**: They form an indefinite loop of redirections. For example, `HTTP://www.abc.com/first/second/first/second/...`.



Classification of web crawler traps

Mostly, the crawler traps are unintended because of the poor structuring of the website.

Interestingly, crawler traps can also be placed intentionally to dynamically generate an endless series of web pages with the pure purpose of exhausting a crawler's bandwidth. These traps severely impact the throughput of a crawler and hamper its performance.

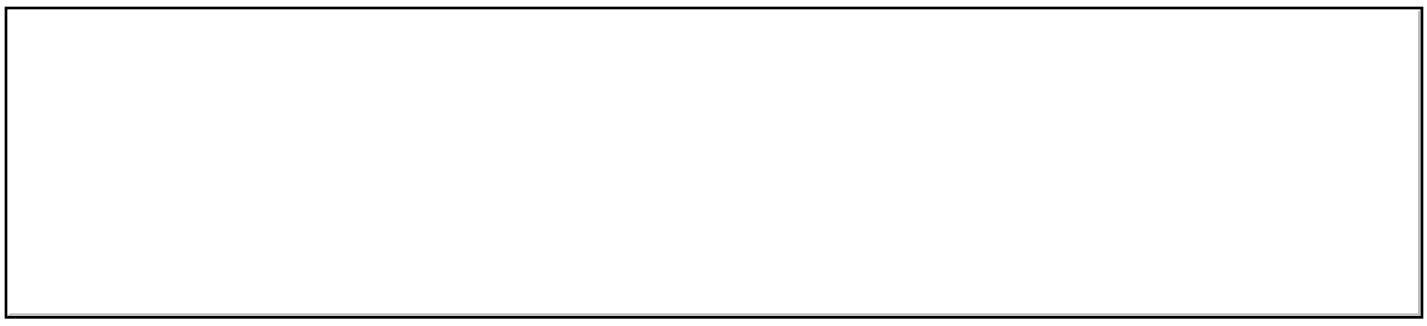
These crawler traps might be detrimental to the crawler, but they also seriously affect the website's SEO ranking.

Identification#

It's essential to categorize the crawling traps to correctly identify them and think of design adjustments accordingly.

There are mainly two layers to the process of identifying a crawler trap:

1. **Analyzing the URL scheme**: The URLs with a poor URL structure, for example, those with cyclic directories: `HTTP://www.abc.com/first/second/first/second/...` will create crawler traps. Hence, filtering such URLs beforehand will rescue our crawler resources.
2. **Analyzing the total number of web pages against a domain**: An impossibly large number of web pages against a domain in the URL frontier is a strong indicator of a crawler trap. So, limiting the crawl at such a domain will be of utmost importance.



The identification process of web crawler traps

Solution#

Our design lacks details on the responsible crawling mechanism to avoid crawler traps

for the identifications described above.

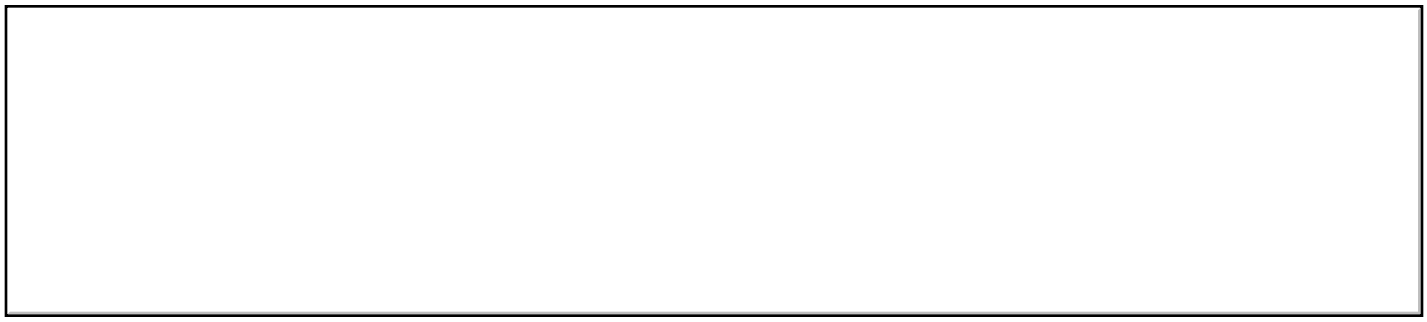
Crawling is a resource and time-consuming task, and it's of extreme importance to efficiently avoid crawler traps in order to achieve timely and helpful crawling. Once the crawler starts communicating with the server for downloading the web page content, there are multiple factors that it needs to take into consideration, mainly at the HTML-fetcher level. Let's go over these factors individually.

1. The crawler must implement an application-layer logic to counter the crawler traps. This logic might be based on the observed number of web pages exceeding a specified threshold.

   The crawler must be intelligent enough to limit its crawling at a specific domain after a finite amount of time or web page visits. A crawler should smartly exit a web page and store that URL as a no-go area for future traversals to ensure performance effectiveness.

2. When initiating communication with the web server, the crawler needs to fetch a file named `robots.txt`. This file contains the dos and don'ts for a crawler listed by the web masters. Adhering to this document is an integral part of the crawling process. It also allows the crawler to access certain domain-specified web pages prioritized for crawling, without limiting its access to specific web pages.

   Another essential component of this document is the revisit frequency instructions for the crawler. A popular website might demand a frequent revisit, contrary to a website that rarely updates its content. This standard of websites communicating with the web crawlers is called the **Robots Exclusion Protocol**. This protocol prevents the crawlers from unnecessarily spending crawling resources on uncrawlable web pages.



The robots exclusion protocol

> The `robots.txt` file doesn't protect crawlers from malicious or intended crawler traps. The other explained mechanisms must handle those traps.

3. Since each domain has a limited incoming and outgoing bandwidth allocation, the crawler needs to be polite enough to limit its crawling at a specific domain. Instead of having a static crawl speed for every domain, a better approach is to adjust the crawl speed based on a domain's **Time to First Byte** (**TTFB**) value. The higher the TTFB value, the slower the server. And so, crawling that domain too fast might lead to more time-out requests and incomplete crawling.

These modifications in the design will ensure a crawler capable of avoiding crawler traps and hence optimizing resources' usage.

Design of a Web Crawler

Next

Evaluation of Web Crawler's Design

Mark as Completed

Report an Issue