Web Crawler

Web crawler is a system for downloading, storing, and analyzing web pages. It performs tasks of organizing web pages that allows users to easily find information.

Use Cases:

- Search Engine Indexing
- Web archiving
- Copyright & Trademark violation monitoring

Requirements:

A simple web crawler should have following functionalities:

- Given a set of URLs (Seed URL), visit the URL, extract metadata of interest such as Title, description, Type, Author etc.
- Web crawler should be robust to handle various challenges, like poorly formatted HTML, unresponsive servers, crashes, malicious links etc.
- Web crawler should respect politeness i.e., they should avoid making too making too many calls to a website within a short time, so that it doesn't lead to DDoS attacks.
- System needs to be modular enough to adapt to any changes in future.
- Web crawler should be manageable and reconfigurable and have good interface for monitoring and statistics.

Estimation

Let's assume, we have 1 billion URLs to crawl every month.

- Average no. of pages we need to query = 1 billion / 30 days / 24 hours / 3600 sec = 400 pages/sec.
- Assuming peak value of pages per second = 2 * Average = 800 pages/sec.
- Assuming average page size = 200 Kb,

  Storage needed per month = 1 billion * 200 Kb = 200 TB per month.
- Assuming we need to store the data for 2 years, before archiving it to cheaper and infrequent data storage, total storage needed = 2 years * 12 months * 200 TB = 4.8 PB.

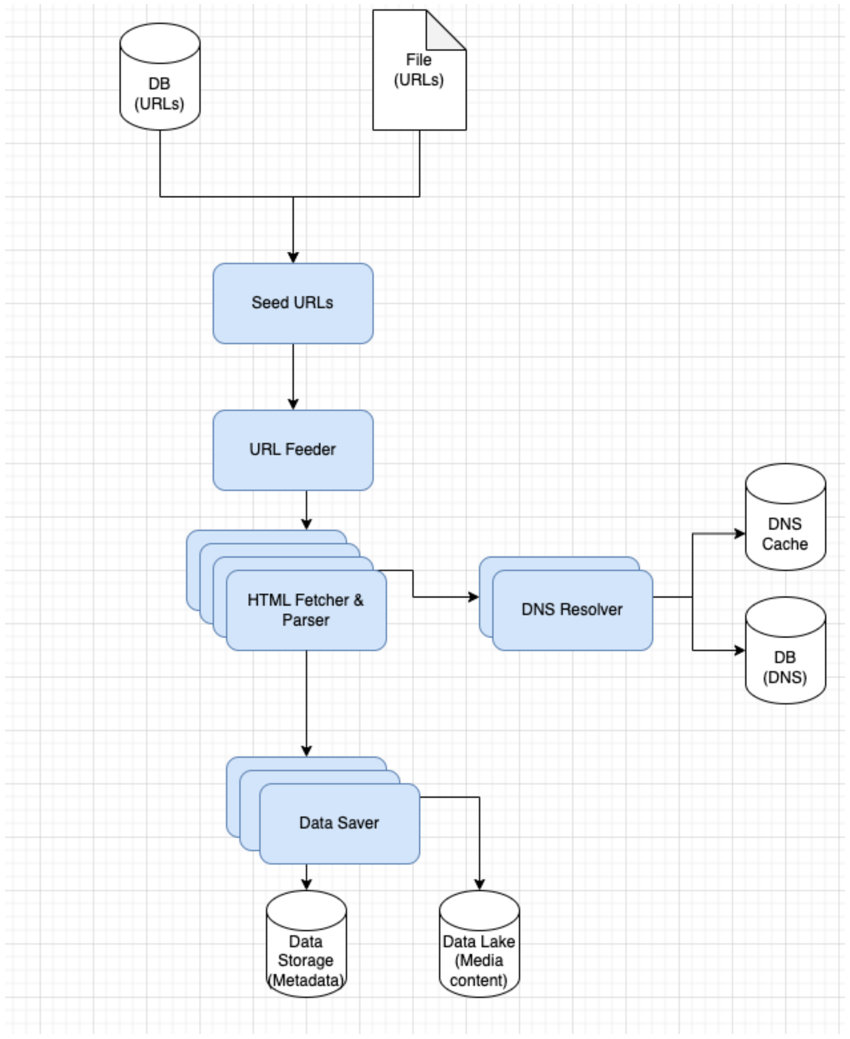
Assumptions

Considering following assumption while designing the crawler:

- Crawling is done as anonymous user, it means, we won't be crawling any page where login is needed.
- All the URLs present in the list (Seed URL) are pre-validated & filtered, and do not contain any malicious content

High Level Design

High level architecture of the web crawler will look like this:



<u>Seed URLs:</u> We can fetch the list of seed URLs from a File or a database (MySQL/PostgreSQL). There are ways to collect seed URLs from internet, which is out of scope for now. We can look at it later.

<u>URL Feeder:</u> It is one of the core components of a Web crawler, which stores the URLs that needs to be crawled. To implement this, we can use one or multiple FIFO (First In First Out)

Queues, where URLs can be processes in the order they were added to the queue. One of solution could be AWS SQS, which is infinitely scalable, and highly reliable.

HTML Fetcher & Parser: It is a component responsible for downloading pages for the given URL (while respecting robots.txt), parse it and return required information. It can be a simple program which returns interested metadata from a given webpage, assuming URL is pre-filtered.

DNS Resolver: For downloading a webpage, URLs must be translated into an IP Address, usually it happens fast, and is handled automatically. But when we have billions of URLs to crawl, it can add a significant overhead. We can resolve this by creating our own local DNS resolver, which could be a cache, where we store domain name and IPs. Since a lot of URLs can belong to same Domain, it can help us in significantly reducing the processing time.

Data Saver: After web pages are downloaded and parsed, they need to be stored in a storage system. But saving large amount of data continuously can sometime cause our database system to crash, and we don't want any loss of data. We can build a Data saver component which implements a FIFO queue and controls the speed by which data is saved in the database. Depending on the type of database, we can implement single or multiple FIFO queues to store data efficiently.

Data Storage: We need to carefully choose the database/storage system to save the parsed data according to our requirement, and future enhancement possibilities.

- If we only want to store only the metadata, we can use NoSQL DBs like MongoDB, since the metadata may differ a lot depending on websites, it will be schema-less, and it supports sharding as well, so that we can efficiently store and query the data as needed.
- If we want to store only interested & fixed metadata such as title, description, type, author etc. we can use MySQL as well, which supports high read and availability.
- If we want to store media content such as images and videos as well, we can compress and store the data in low-cost cloud storage providers like AWS S3.
- If we want to store the content for real-time search or query, we can store the data in large, distributed database such as HDFS, Google Big Table, Apache Cassandra etc., which are designed to support querying and searching.

URL Cache: Since we have billions of URLs to crawl, we can have a lot of duplicate URLs as well (Depending on how the seed URL list was generated). To improve the efficiency of our web crawler, we can use a cache to store recently processed URLs, and quickly look for a URL in the cache before crawling it again.

URL Filter (optional): There may be times, where we have a lot of URLs which we do not want to crawl, eg. Websites we aren't interested in, malicious websites, faulty links, websites containing pornographic materials etc. We can build a component to validate the URL and filter it before crawling. We can improve the efficiency and effectiveness of our web crawler by limiting the amount of unnecessary or irrelevant content that we need to crawl.

Workflow

==

Insert workflow Diagram Here

==

The crawling process of our web crawler will consist of multiple servers that perform repeated cycles of works parallelly. Below is the summary of steps involved:

1. We start by fetching the list of seed URLs from a database or given text file. We need to send these URLs to the URL Feeder, which is responsible for providing URLs to HTML Fetcher, while respecting the priority and politeness policy of a URL.

Low Level Design

URL Feeder:

==

Insert Low Level Diagram of URL Feeder Here

==

Optimizations

Below is the list of further optimizations we can do to further enhance our system:

- We can divide HTML Fetcher & Parser in two components HTML Fetcher & HTML Parser, and add another component for duplicate detection, so that we do not have to parse a content, which is already parsed.
    - We can save a hash of content for each URL, and compare
-

Planning & Next Steps

We can divide the engineering to team into multiple teams, each of them can handle one or more of below components:

1. URL Feeder
2. HTML Fetcher & Parser
3. Data Saver