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:

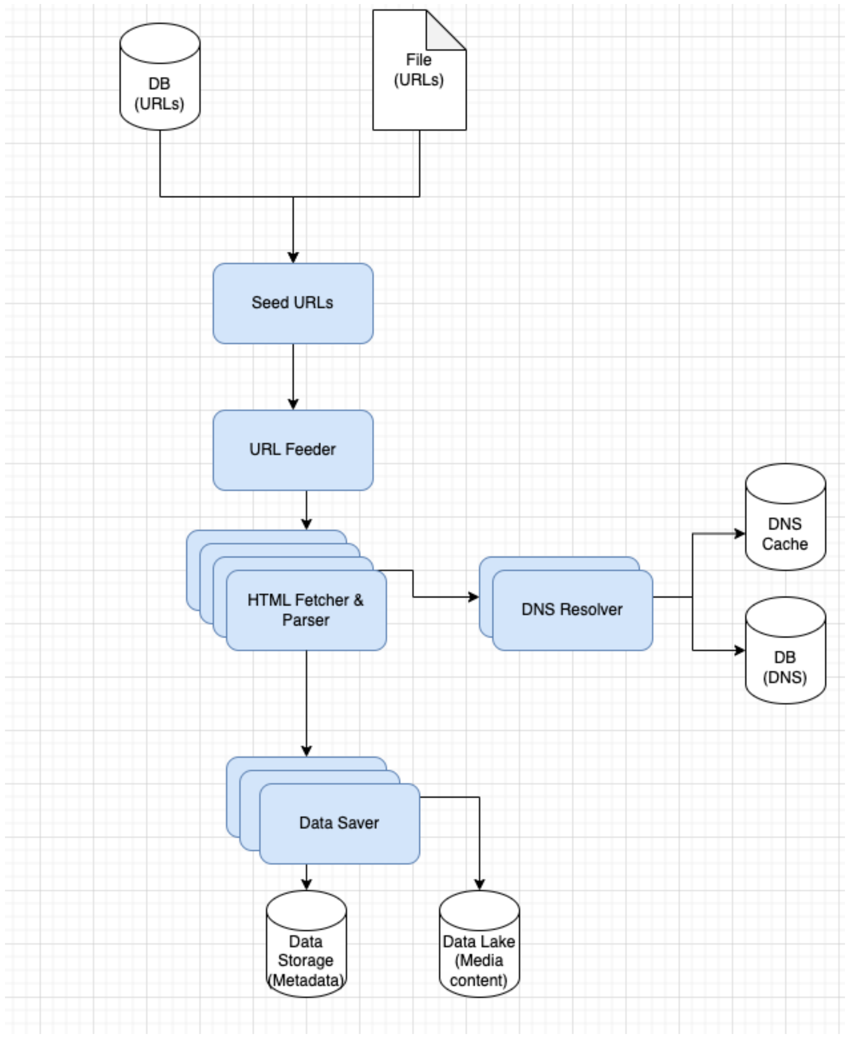

Figure representing High level design

Seed URLs: 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.

URL Feeder: 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

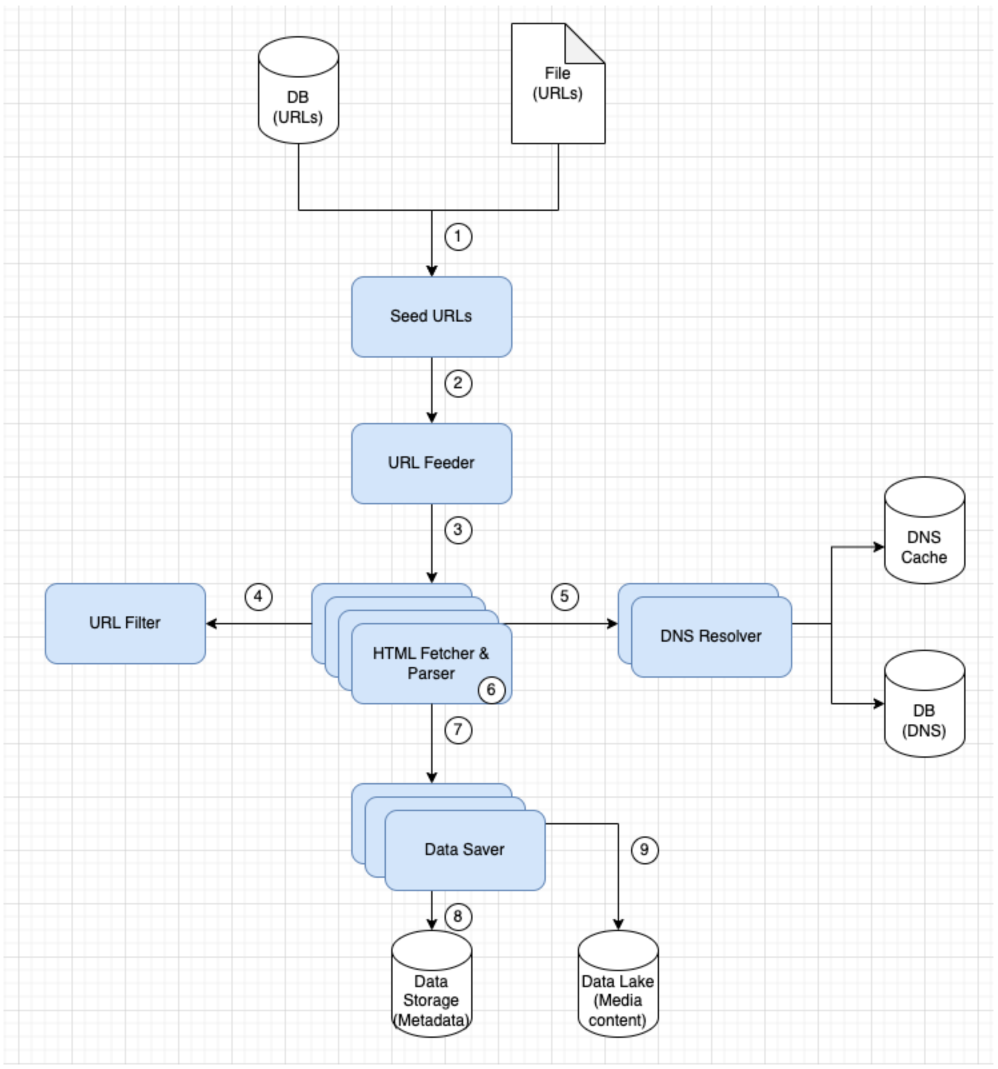

Figure representing Workflow

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.
2. We need to send seed URLs to the URL Feeder, which is responsible for providing URLs to HTML Fetcher, while respecting the priority and politeness policy of a URL.
3. HTML Fetcher and parser module, fetches URL from URL Feeder.
4. HTML Fetcher calls URL Filter module to validate URL before crawling.

5. HTML Fetcher module calls DNS Resolver module to fetch the IP address of the domain associated with the URL.
6. HTML Parser retrieves and parses the HTML page, and extracts required data.
7. HTML Parser sends the parsed data to Data saver module
8. Data saver saves text data (metadata) to the Database (according to our requirement).
9. Data saver saves media content in low-cost storage system like AWS S3.

Low Level Design

URL Feeder:

URL Feeder is responsible for giving the URLs to N number of servers which are running HTML Fetcher & Parser module. It has to give priority to a URL (if needed) and also respect politeness i.e. we need to have only 1 connection from crawler to a given webserver at a time, and there also needs to be a delay between subsequent calls to a webserver.

To achieve this, we can have multiple submodules in the URL feeder as below:

- URL Prioritizer
    - Gives priority from 1 to M, to each URL based on past data or some algorithm. It may be based on interests or how frequently content changes in a website.
    - It inserts URL with priority 1 in Front Queue #1, URL with priority 2 in Front Queue #2 and so on.
- Front FIFO Queues
    - Maintains a queue of URLs based on their priority, where higher number means higher priority.
- Back Queue Feeder
    - It has the job to make sure no Back Queues are empty, and all the URLs from one domain are sent to one Back Queue only. One Back queue can have URLs from multiple domains, but same domain URLs can’t be in multiple Queues
    - It can use a cache or DB to store and figure out URLs from which domain resides in which Back Queue.
    - The number of back queues should match the number of parallel servers running HTML Fetcher & Parser module.
- Back FIFO Queues
    - Maintains a queue of URLs based on their domain.
- URL Selector
    - Fetches the next URL for a given server number and passes it to them, so that all the URL of same domain goes to same server, and all subsequent requests have some time delay in between them.

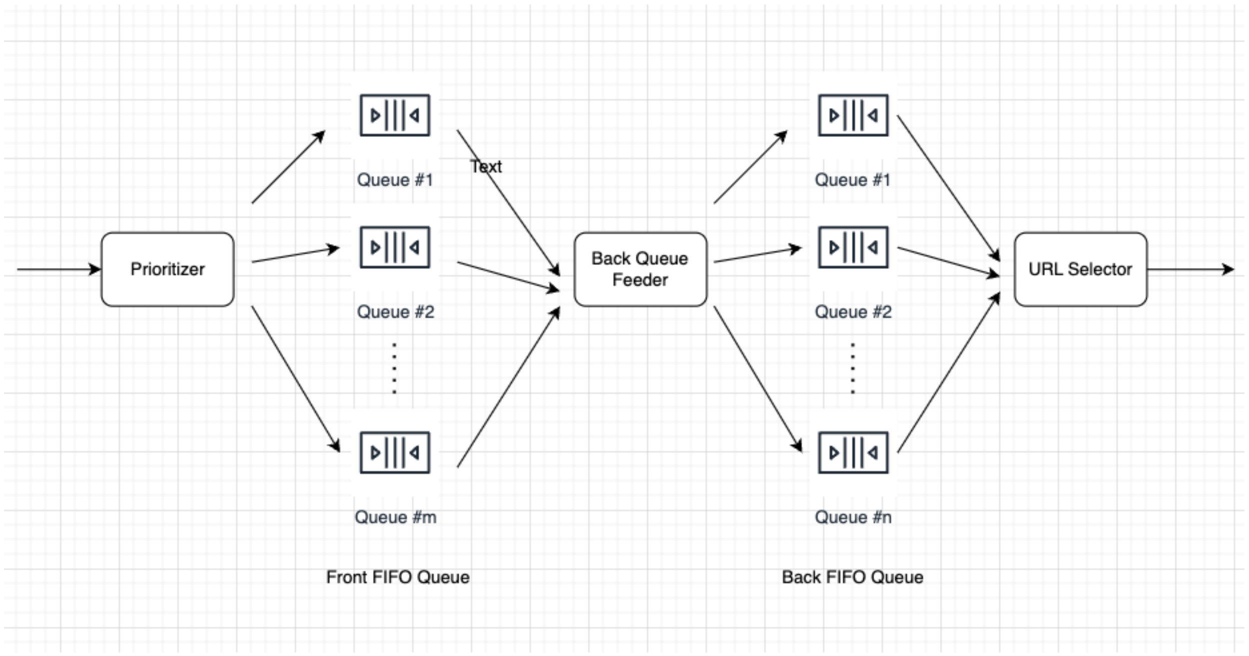

Fig representing URL Feeder

HTML Fetcher & Parser

HTML Fetcher & Parser needs to fetch data from webservers while respecting robots.txt and following politeness. HTML Fetcher & Parser module can be divided into multiple modules as below, and uses DNS Resolver and URL Filter module to fetch and parse data efficiently.

Below workflow defines how HTML Fetcher & Parser module works internally:

- HTML Fetcher fetches URL to crawl from URL Feeder
- It then validates the URL with URL filter module, whether it should be crawled or not.
- For any URL that passes URL filter, HTML Fetcher calls DNS Resolver to fetch IP address of given domain.
- It strips all the trailing slash from the path and adds robots.txt to it and fetches exclusion lists of given webserver.
- Every path in exclusion list is added to URL filter, so that it is not crawled by the web crawler.
- If current path is in exclusion list, it is not crawled.
- HTML Parser then retrieves webpage and parses required content.

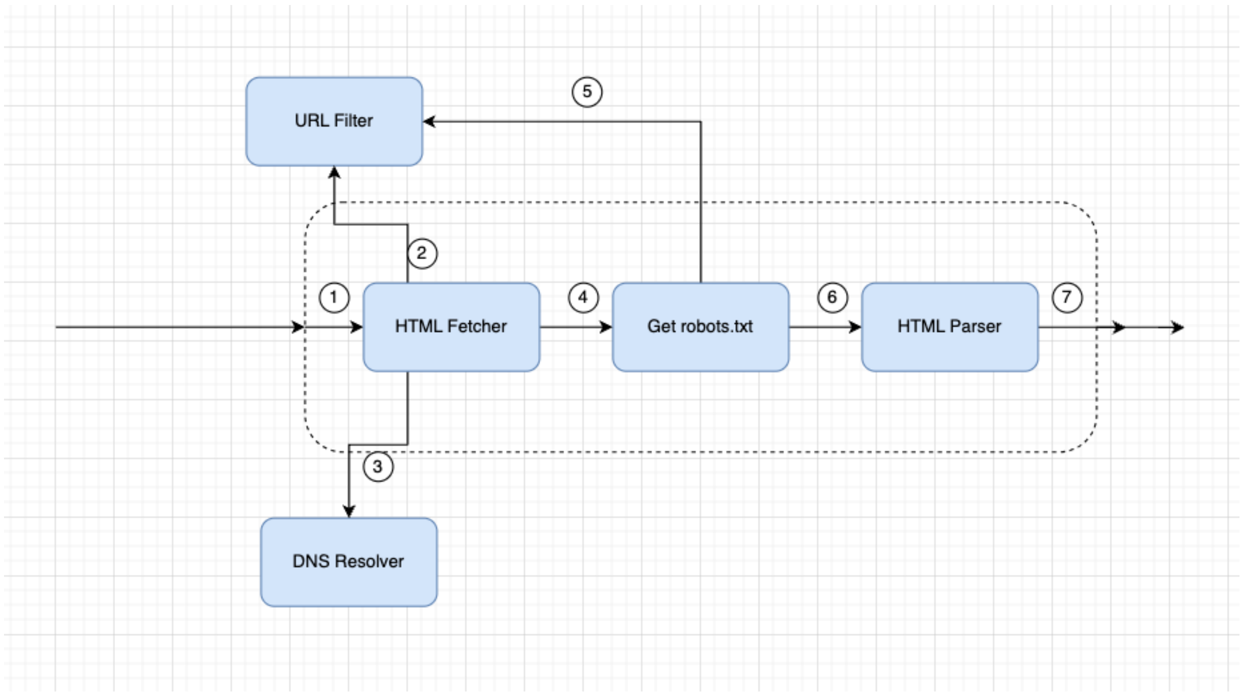

Fig representing HTML Fetcher & Parser

Potential Blockers and Pitfalls

- To process billions of requests, we need to have really high number of servers running in parallel. Handling large fleet of distributed servers running parallely will be cumbersome.
- If a website has relatively much large number of URLs to crawl, can put loan on one of the servers running HTML Fetcher & Parser module, as we are trying to fetch & parse all URLs of same domain from same server.
- FIFO Queues needs to proper eviction after processing of a URL is done, otherwise it may get processed multiple times.
- Since the number of back FIFO servers (URL Feeder module) must match with the number of servers running HTML Fetcher & Parser module, if either one of them fails, we need to have a backup mechanism to distribute the load in remaining set of servers. We can utilize consistent hashing for this.
- Testing a system which can crawl billions of URLs will be very challenging, and needs mocks representing all sorts of scenarios.
- Poorly formatted HTML pages can have the data, but we may not be able to retrieve unless we do some custom coding
- Dynamically adding more servers to fetch and parser webpage will need addition of back queues, and rehashing of domain names so that they go to same servers again (not necessarily same as before)

Planning & Next Steps

We can divide the engineering team into 5 teams, each of them can handle below components as divided below:

1. URL collection & adding it to URL feeder
2. URL Feeder
3. HTML Fetcher & Parser
4. DNS Resolver, URL Filter
5. Data Saver

The overall timeline can be divided into following phases, and we can monitor the progress of each phase to reach our milestones in planned manner.

- Phase 1
  - Creating stories for each module and submodule.
  - Creating subtask for each module which can be completed in one sprint.
  - Assigning modules to each team, and further working with team leads to further assign subtasks to individual members.
- Phase 2
  - Development of individual modules by respective engineering teams as divided above
    - URL Collection
    - URL Feeder
    - HTML Fetcher
    - HTML Parser
    - DNS Resolver
    - URL Filter
    - Data Saver
  - Depending on the team size, one team might be handling multiple modules, and hence we need to plan so that dependency between teams is less.
- Phase 3
  - Testing of individual modules
  - Deployment of individual module on cloud
  - Performing load testing for 1/10 of load. In this case, we can start with 1/100 or 1/1000 of actual load since we are testing for billions of URLs.
  - Depending on the number of QA teams, we may need more time to complete this phase of initial testing.
- Phase 4
  - Incorporating feedback from initial testing
  - Re-testing as needed.

- Phase 5
    - Integration of all the different modules
    - Basic integration tests by dev team to validate the working of overall functionality
- Phase 6
    - Testing of overall system
    - Deployment of all the modules in cloud
    - Performing load tests for 1/10 of actual load
- Phase 7
    - Incorporating feedback from integration tests
    - Re-testing as needed
- Phase 8
    - Final testing for 1/5th of actual load
    - Deployment to production system
- Phase 9
    - Monitoring of production system to validate that its working as intended.
    - Tweaking production system to improve performance and efficiency.