Interview Daemon

System Design: Web Crawler

Interview Daemon Jan 2 · 6 min read

Source: *interviewdaemon.com* *(Link)*

A Web Crawler sometimes called a **spider** or **spiderbot** and often shortened to **crawler**, is an Internet bot that systematically browses the World Wide Web, typically for the purpose of Web indexing.

Introduction

The most popular example is that Google is using crawlers to collect information from all websites. Besides search engine, news websites need crawlers to aggregate data sources. It seems that whenever you want to aggregate a large amount of information, you may consider using crawlers. (more..)

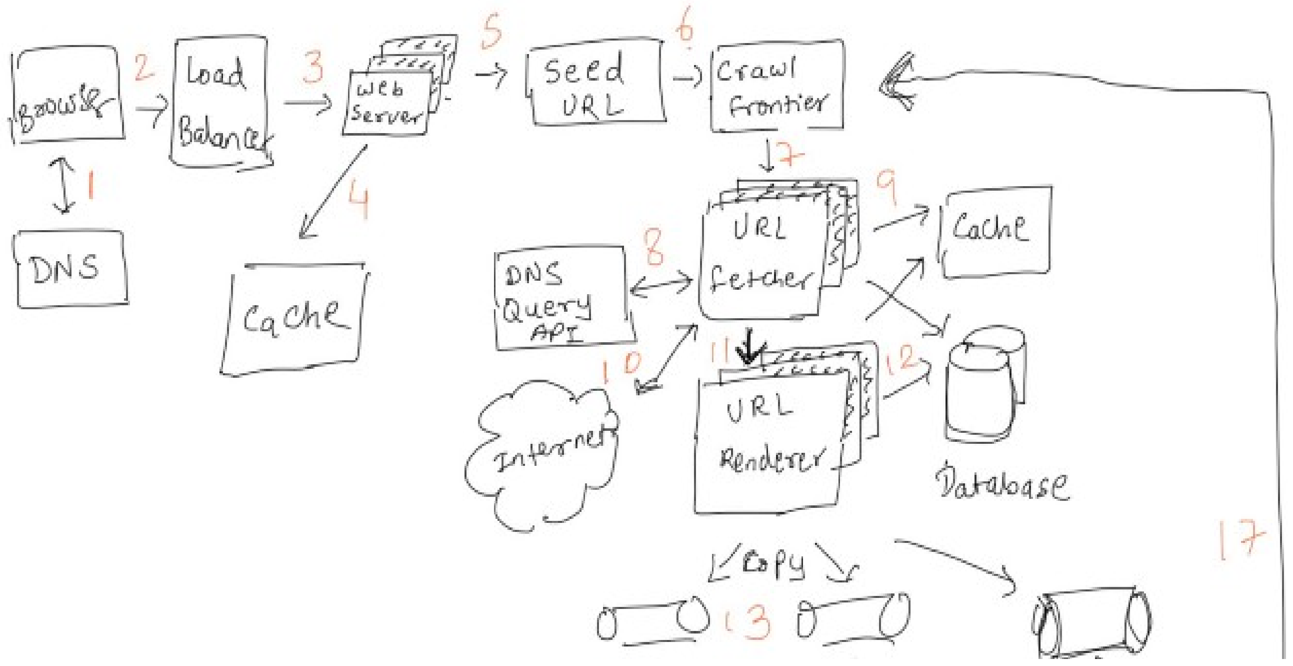
Requirements

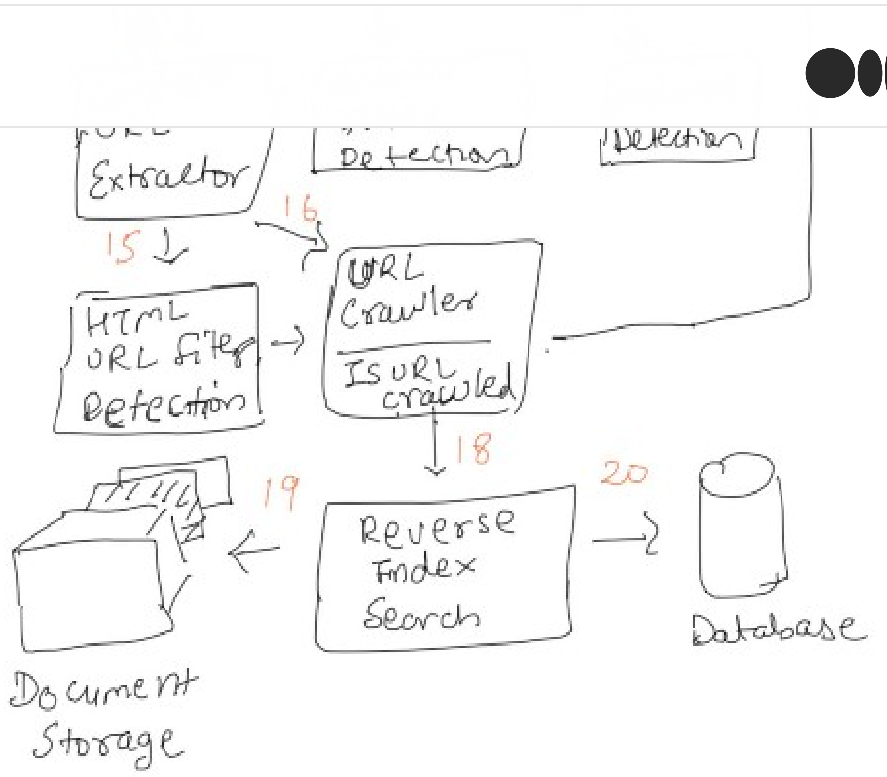
- Service should be able to **crawl the given url**
- Generate **title and page snippets** for each pages that are crawled

- We are only considering crawling html content of web pages. We are currently not **considering mime, image or any media types**.
- Considering Robots Exclusion Protocol (**robots.txt**) for websites that do not want to crawl certain webpages.
- **Minimize the intercommunication** between machines.
- System should be **highly available**
- System should be **highly reliable**
- Service will be both **write-heavy and read-heavy**.
- Traffic may not be evenly distributed as in one website may have 1000 hyperlinks or child webpages(subdomains or pages) while other website may have 2–5 webpages.
- We are NOT considering Page Ranking Algorithm
- Crawler should not get stuck in infinite loops.

High Level System Design

At a high-level, we need to support multiple micro-services in the current system.





Source: Interview Daemon (Link)

API Design

We can use REST or SOAP to serve our APIs. Basically, there will be three important API's of photo and video sharing service system.

1. crawlUrl

```
crawlUrl(api_dev_key, url, media_type)
```

crawlUrl will responsible for crawling the url and all the other subpages inside the url. api_dev_key is the API developer key of a registered account. We can eliminate hacker attacks with api_dev_key. This API returns HTTP response. (202 accepted if success)

2. searchWord

```
searchWord(api_dev_key, search_query, page, maximum_page_count = 20)
```

Database / Storage

NoSQL Database.

- Datastore is used at multiple places for multiple purpose.
- Storing list of pages that are crawled and needs to be crawled with timstamp which will be used by URL Frontier
- We can use data store to store the url and its meta data.
- Storing Reverse Search Index for text and its related url, title and snippet link also with the location of the document store that has the parsed url and its related details.

Document Store (S3)

- That contains the actual document content of the website. Which can later be used to process map reduce for further reverse indexing of the content and for analytics.

Component Design

1. Seed URL

A web crawler starts with a URL to visit, called the *seeds*.

2. Crawl Frontier (Url Frontier)

URL Frontier is a data structure that contains all the URLs that are to be downloaded to crawl. Calling the crawl frontier will return the next URL that we need to be crawled.

DFS vs BFS for Crawling

- We can crawl sub URL's by DFS (depth first search) or BFS (breadth first search) starting from root page and traverse inside. BFS with queue is recommended here so that we can keep adding the URL's to the queue as FIFO (First in First Out).

Distribute using Hash

which is responsible for crawling it.

Politness

- Crawler should not overload a server by downloading lots of pages
- Each worker thread will have its separate sub-queue, from which it removes URLs for crawling. When a new URL needs to be added, the FIFO sub-queue in which it is placed will be determined by the URL's canonical hostname.
- Should not have multiple machines connecting to WebServer
- Hash function will map each hostname to a thread number

3. URL Fetcher

URL Fetcher purpose is given a URL from URL Frontier identify the DNS and later fetch the ip and pass it to the URL Renderer. HEre we can also validate against the robot.txt to make certain parts of the website's are not crawled as per the robots.txt.

4. Custom DNS Server

DNS name resolution will be a big bottleneck of our crawlers given the amount of URLs we will be working with. To avoid repeated requests, we can start caching DNS results by building our local DNS server.

5. URL Renderer

URL Renderer download's the web document corresponding to a given URL using the appropriate network protocol like HTTP (Here we can discuss on if we have to use any other protocol like FTP). Here instead of downloading a document multiple times, we **Cache** the document locally.

6. Content Extractor (Document Stream Reader)

We can use an input stream reader that reads the entire content of the document which was downloaded from the URL Renderer and also can provide a method to re-read the same document if we pass the same url. The Content Extractor extracts the content and passes the content (Rabbit MQ or Kafka) to **Content Filter Detection** and **URL Filter Detection** not want to store in our data store. Some potential algorithms are Jaccard index and cosine similarity. Here we are only filtering text content from the docment.

8. URL Filter Detection (URL De-dupe)

Similar to Content Filtering URL filter is filtering he URL's from the given document. We need to be careful the web crawler doesn't get stuck in an infinite loop, which happens when the graph contains a cycle. This can also be used to blacklist URL's. We can pass these filtered URL to URL Crawler which intern passes the URL to URL Frontier for next run. Here we can avoid URL's which has mime objects like interviewdaemon.com/welcome.png will be filtered out and wont be traversed further.

9. URL Crawler

10. Reverse Index Search

Based on the document content. Reverse Index Search will remove markup breaks up the text into multiple terms. It also normalizes the content and converts the query to use boolean operators. Once we have the query it can be used to retrieve the page and snippet given a text.

The **Reverse Index Service** ranks the matching results and returns the top ones.

```
curl https://search.com/api/v1/search?query=hello+world
```

Determining when to update the crawl results

Pages need to be crawled regularly to ensure freshness. Crawl results could have a `timestamp` field that indicates the last time a page was crawled. After a default time period, say one week, all pages should be refreshed. Frequently updated or more popular sites could be refreshed in shorter intervals.

Data Sharding (Data Partitioning)

Our crawler will be dealing with three kinds of data:

1. URLs to visit

Since we are distributing URLs based on the hostnames, we can store these data on the same host. So, each host will store its set of URLs that need to be visited, checksums of all the previously visited URLs, and checksums of all the downloaded documents. Since we will be using consistent hashing, we can assume that URLs will be redistributed from overloaded hosts.

Cache and Load Balancing

We can introduce a cache for metadata servers to cache hot database rows. Least Recently Used (LRU) can be a reasonable cache eviction policy for our system.

How can we build a more intelligent cache? If we go with the eighty-twenty rule, i.e., 20% of daily read volume for URLS is generating 80% of the traffic, which means that certain URL's are so popular so that most people search them. This dictates that we can try caching 20% of the daily read volume.

References:

- InterviewDaemon
- DonneMartin
- Wikipedia
- TechDummies

System Design Interview System Design Interview Daemon